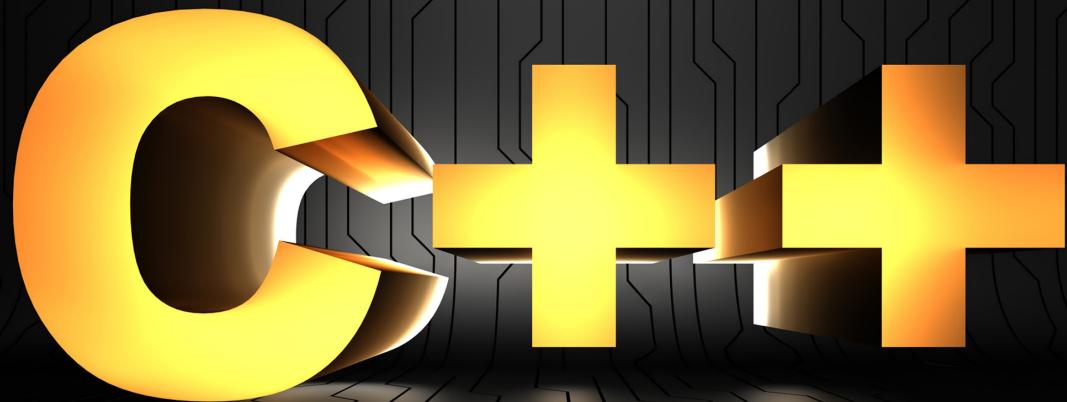


PROGRAMMING IN



Prof. Partha Pratim Das

Computer Science & Engineering

Indian Institute of Technology, Kharagpur



INDEX

S.No	Topic	Page No.
	<i>Week 1</i>	
1	Recap of C	01
2	Recap of C	15
3	Recap of C	28
4	Programs with IO and Loop	46
5	Arrays and Strings	59
6	Sorting and Searching (Lecture 06)	71
7	Stack and its Applications	83
	<i>Week 2</i>	
8	Constants and Inline Functions	95
9	Constants and Inline Functions (Contd.)	114
10	Reference and Pointer	127
11	Reference and Pointer (Contd.)	139
12	Default Parameters and Function Overloading	148
13	Default Parameters and Function Overloading (Contd.)	162
14	Default Parameters and Function Overloading (Contd.)	173
15	Operator Overloading	186
16	Operator Overloading (Contd.)	199
17	Dynamic Memory Management	208
18	Dynamic Memory Management (Contd.)	221
	<i>Week 3</i>	
19	Classes and Objects	229
20	Classes and Objects (Contd.)	246
21	Access Specifiers	251
22	Access Specifiers (Contd.)	266
23	Constructors, Destructors and Object Lifetime	271

24	Constructors, Destructors and Object Lifetime (Contd.)	285
25	Constructors, Destructors and Object Lifetime (Contd.)	294
26	Copy Constructor and Copy Assignment Operator	308
27	Copy Constructor and Copy Assignment Operator (Contd.)	324
28	Copy Constructor and Copy Assignment Operator (Contd.)	338
29	Const-ness	345
30	Const-ness (Contd.)	366

Week 4

31	Static Members	375
32	friend Function and friend Class	391
33	Overloading Operator for User Defined Types: Part - I	404
34	Overloading Operator for User Defined Types: Part - II	419
35	Namespace	436

Week 5

36	Inheirtnace : Part I	455
37	Inheritance : Part II	468
38	Inheritance : Part III	481
39	Inheritance : Part IV	495
40	Inheritance : Part V	515

Week 6

41	Dynamic Binding : Part I	534
42	Dynamic Binding (Polymorphism) : Part II	547
43	Dynamic Binding (Polymorphism) : Part III	564
44	Dynamic Binding (Polymorphism) Part IV	585
45	Dynamic Binding (Polymorphism) : Part V	600

Week 7

46	Virtual Function Table	619
47	Type casting and cast operators : Part I	636
48	Type casting and cast operators : Part II	661
49	Type casting and cast operators : Part III	679
50	Multiple Inheritance	697
51	Multiple Inheritance (Contd.)	714

Week 8

52	Exceptions (Error Handling in C) : Part I	726
53	Exceptions (Error Handling in C) : Part II	753
54	Template (Function Template) : Part I	772
55	Template (Function Template) : Part II	791
56	Closing Comments	809

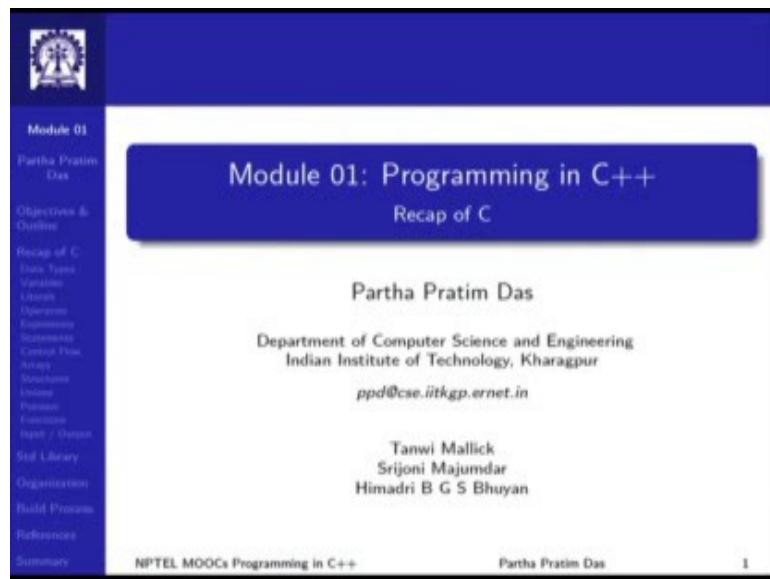
Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 01
Recap of C (Part I)

Welcome to programming in C++. This will be 20 hour course, where we will talk about various aspects of the C++ programming language and it will be divided naturally into about 40 modules that you will study one after the other.

The main emphasis of this course is to teach, how C++ programming language should be used in designing and implementing complex software systems.

(Refer Slide Time: 00:57)



So, you all will be aware that C++ is object oriented or object based programming language and I would assume that you know C language, may not be at a very depth, but you have the overall idea about the C language. So, we will start from there, in the module 1, we will primarily talk about recapitulating various specific aspects of C programming. This is just to make sure that you can, if required you can revisit those concepts and before we get deep into the C++ language, you can be familiar with all the

C programming requirements because C is a language, which is backward compatible to C++. So, we will first get started with recapitulation of C.

(Refer Slide Time: 01:56)

The slide is titled "Module Objectives". It features a blue header bar with the title and a logo. The main content area contains a bulleted list of objectives. On the left, there is a sidebar with navigation links for the module. At the bottom, there is footer information.

Module Objectives

- Revisit the concepts of C language
- Revisit C Standard Library components
- Revisit the Organization and Build Process for C programs
- Create the foundation for the concepts of C++ with backward compatibility to C

Module 01
Partha Pratim Das

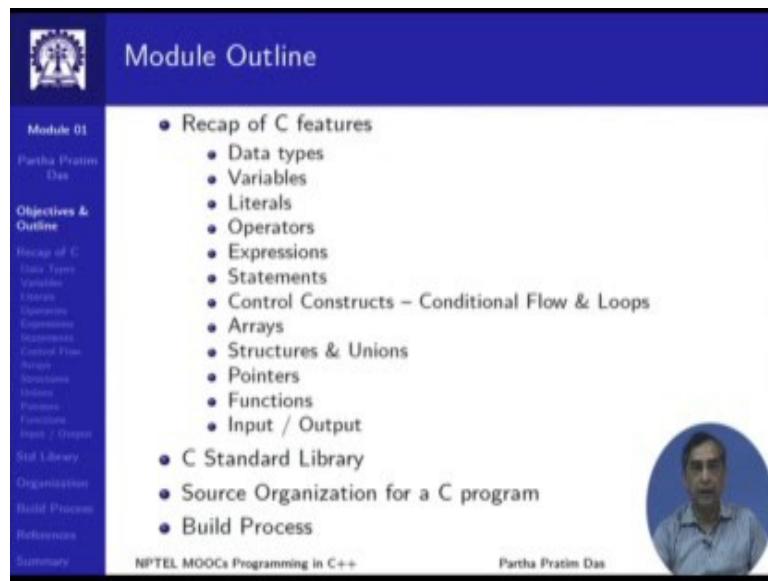
Objectives & Outline

- Basics of C
- Data Types
- Variables
- Control
- Operators
- Expressions
- Statements
- Control Flow
- Arrays
- Functions
- Pointers
- Functions
- Type / Scope
- Std Library
- Organization
- Build Process
- References
- Summary

NPTEL MOOCs Programming in C++ Partha Pratim Das 2

So, these are objectives to revisit the concepts, particularly we will look into C Standard Library, besides the C language and programming aspects. We will briefly discuss about the organization of C program, how C program is to be organized possibly. So, far you have only written code in terms of one single file using possibly 1 or 2 functions only. One of them must be main, as you know, we will show how to organize programs better and with this we will have a foundation to; for the C++ programming language.

(Refer Slide Time: 02:36)



This slide shows the 'Module Outline' for NPTEL MOOCs Programming in C++. It features a sidebar on the left with navigation links and a main content area on the right.

Module Outline

Module 01
Partha Pratim Das

Objectives & Outline

Recap of C
Data Types
Variables
Operators
Expressions
Statements
Control Flow
Arrays
Structures
Unions
Pointers
Functions
Input / Output

Std Library
Organization
Build Process
References
Summary

NPTEL MOOCs Programming in C++ Partha Pratim Das

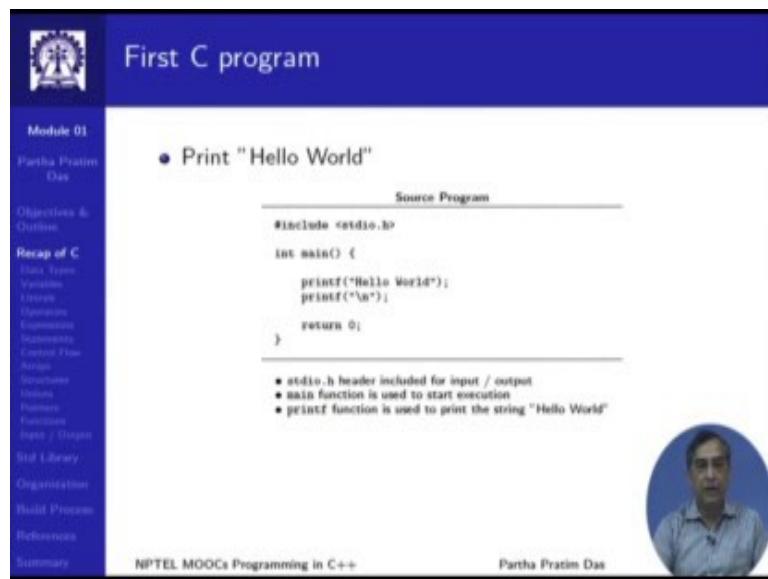
The main content area lists the module outline:

- Recap of C features
 - Data types
 - Variables
 - Literals
 - Operators
 - Expressions
 - Statements
 - Control Constructs – Conditional Flow & Loops
 - Arrays
 - Structures & Unions
 - Pointers
 - Functions
 - Input / Output
- C Standard Library
- Source Organization for a C program
- Build Process

A circular profile picture of Partha Pratim Das is located in the bottom right corner of the slide.

These are the different topics to be done, the outline of the module; this is for reference. As the presentation, we will proceed on the left of your screen, you will see this outline and it will be highlighted as to which particular topic we are talking about.

(Refer Slide Time: 02:54)



This slide shows the 'First C program' section of the NPTEL MOOCs Programming in C++ course.

First C program

Module 01
Partha Pratim Das

Objectives & Outline

Recap of C
Data Types
Variables
Operators
Expressions
Statements
Control Flow
Arrays
Structures
Unions
Pointers
Functions
Input / Output

Std Library
Organization
Build Process
References
Summary

NPTEL MOOCs Programming in C++ Partha Pratim Das

The main content area highlights the task of printing "Hello World".

- Print "Hello World"

Source Program

```
#include <stdio.h>
int main() {
    printf("Hello World");
    printf("\n");
    return 0;
}
```

Annotations:

- stdio.h header included for input / output
- main function is used to start execution
- printf function is used to print the string "Hello World"

A circular profile picture of Partha Pratim Das is located in the bottom right corner of the slide.

So, this is the first program “Hello World” which I am sure all of you have studied. This is also the starting program in Kerning and Ritchie’s famous book. We use ‘printf’ from the <stdio> library and print the hello world on to the terminal or which is formally set to with the <stdio> out file. The main function is one that you can see here is where the execution starts and then you print this string and print ‘\ n’, which means you basically go to the next line; new line.

(Refer Slide Time: 03:37)

The slide has a blue header with the title 'Data Types'. On the left, there's a sidebar with a logo and a navigation menu. The main content area contains text and bullet points about C data types.

Data Types

Data types in C are used for declaring variables and deciding on storage and computations:

- **Built-in / Basic** data types are used to define raw data
 - char
 - int
 - float
 - double

Additionally, C99 defines:

- bool

All data items of a given type has the same size (in bytes). The size is implementation-defined.

- **Enumerated Type** data are internally of int type and operates on a select subset.

NPTEL MOOCs Programming in C++ Partha Pratim Das 5

C has a number of data types. Those are known as char, which is character; int, float and double; float and double for the floating point numbers and integers are for the so called whole numbers.

Now, here I should mention that the C that you commonly use is known as C89, C89 is a first standard of C that was created by ANSI, the standardization organization and subsequently in 99, another standard was released, this is called C99, so most of the compilers today follows C99 standard. We will also expect C99 to be followed. So, when we talk about C, we will try to highlight, if few things have become different in C99. So, in terms of data type as you can see, there is a new data type bool, which has got added in C99.

In C89, you could still have Boolean values, which can be true or false based on it being an integer value. So, if it is 0, it is false; otherwise it is true. But in C99, there is a separate type `bool`. Every data type as you know, this built-in data types has a size that is given in bytes and you can use ‘`sizeof`’ operator to get that. You can define enumerated types which are basically integer values which are given some symbolic names.

(Refer Slide Time: 05:17)

The slide is titled "Data Types" and is part of "Module 01" by Partha Pratim Das. The content discusses various data types in C, including void, derived data types (arrays, structures, unions, pointers, functions), and type modifiers (short, long, signed, unsigned). The slide also mentions that strings are not a type but can be manipulated using functions from <string.h>. The footer includes the NPTEL MOOCs logo and the slide number 6.

- **void:** The type specifier `void` indicates no type.
- **Derived data types include:**
 - Array
 - Structure – `struct` & `union`
 - Pointer
 - Function
 - String – C-Strings are really not a type; but can be made to behave as such using functions from `<string.h>` in standard library
- **Type modifiers include:**
 - `short`
 - `long`
 - `signed`
 - `unsigned`

Other data types in C include ‘`void`’. ‘`void`’ is not a type, it is very interesting use and as we go into C++, we will see various different use of `void`. `Void` is where you would need to use a type, you can use the type `void`, but it actually says that there is no type. So, it is like, when we do arithmetic we have a 0. So, I can add 0 to `x` and it does not change `x`. So, as we say every system needs a 0. So, `void` is a 0 of the type system, as we will see more in C++.

Then based on this built in types, there are various derived types that supported the array, structure and union, pointer; we can have functions and it is the commonly called, there is a string type in C called C strings these days. Very strictly speaking string is not a type in C, you will understand that more when we go into C++. C strings are actually a collection of functions in `<string.h>` header, which allow us to manipulate strings in C.

Finally, the data types can be modified for their size and whether they will be signed or unsigned and these 4 type modifiers are used in C.

(Refer Slide Time: 06:42)

The screenshot shows a slide titled "Variables" from a C++ programming course. The sidebar on the left contains a logo, the title "Module 01", and a list of topics: Partha Pratim Das, Objectives & Outline, Basics of C, Data Types, Variables, Control, Operators, Expressions, Functions, Loops, Pointers, Functions, Input / Output, Std Library, Organization, Build Process, References, and Summary. The main content area has a blue header "Variables". Below it, there are two bullet points: "A variable is a name given to a storage area" and "Declaration of Variables:". Under "Declaration of Variables", there are two more bullet points: "Each variable in C has a specific type, which determines the size and layout of the storage (memory) for the variable" and "The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore". Below these points is some sample code:

```
int i, j, noOfData;
char c, endOfSession;
float f, velocity;
double d, dist_in_light_years;
```

At the bottom right of the slide is a circular video player showing a man's face, identified as Partha Pratim Das. The footer of the slide includes the text "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

We will move on there are as you know the variables in C. Variables have certain, their names can be defined in certain ways starting with an alpha or an underscore and then extended with alpha numeric.

Here are some examples of different variable names, while it is often convenient to name variables with single letters or 1-2 letters. It is advised that you use variable names which make some meaning. So, we are saying character 'endOfSession', you could have just called it 'c' or 'd' or 'a', but it is better to give it in name from which it can be understood as to what the variable means.

(Refer Slide Time: 07:28)

The screenshot shows a slide from a C++ programming course. The title 'Variables' is at the top. On the left, there's a vertical navigation menu with 'Module 01' and 'Partha Pratim Das'. Below that is a list of topics: Objectives & Outline, Recap of C, Data Types, Variables, Operators, Expressions, Statements, Control Flow, Arrays, Structures, Unions, Pointers, Functions, Input / Output, Std Library, Organization, Build Process, References, and Summary. The main content area has a bullet point 'Initialization of Variables:' followed by a sub-bullet 'Initialization is setting an initial value to a variable at its definition'. Below this is some sample C++ code:

```
int i = 10, j = 20, numberOfWorkDays = 22;
char c = 'x';
float weight = 4.5;
double density = 0.0;
```

In the bottom right corner, there is a circular profile picture of Partha Pratim Das and the text 'Partha Pratim Das'. At the bottom center, it says 'NPTEL MOOCs Programming in C++'.

When the variables are declared as they are declared here, then the variables can be initialized also. That initialization is optional. So, when we say int i initialized with 10. It means that 'i' is an int type variable whose value at the point of definition itself will become 10. So, if you do not give initialization then it is uninitialized variable which will have an unknown value to start with. Certainly, it is very good to initialize all variables that we declare and define.

(Refer Slide Time: 08:05)

The slide has a blue header bar with the title 'Literals'. On the left, there is a vertical sidebar with a logo at the top and a list of navigation links. The main content area contains bullet points and code examples.

Module 01
Partha Pratim Das

Objectives & Outline
Recap of C
Data Types
Variables
Literals
Operators
Expressions
Statements
Control Flow
Arrays
Structures
Unions
Pointers
Functions
Input / Output
Std Library
Organization
Build Process
References
Summary

Literals

- Literals refer to fixed values of a built-in type
- Literals can be of any of the basic data types

```
212    // (int) Decimal literal
0173   // (int) Octal literal
0b1010 // (int) Binary literal
0x92   // (int) Hexadecimal literal
3.14   // (double) Floating-point literal
'g'    // (char) Character literal
"Hello" // (char *) String literal
```

- In C99, literals are constant values having const types as:

```
212    // (const int) Decimal literal
0173   // (const int) Octal literal
0b1010 // (const int) Binary literal
0x92   // (const int) Hexadecimal literal
3.14   // (const double) Floating-point literal
'g'    // (const char) Character literal
"Hello" // (const char *) String literal
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

C has a number of literals which are basically fixed values of built-in types depending on how you write a particular literal, the type of that literal is decided. For example, if you just have a sequence of numerals then it becomes a decimal integer type, but if you prefix that with 0, then it is considered to be an octal type, a base eight number. If you prefix it with 0x, then it is considered to be a hexadecimal literal and so on. Character literals are within single quotes and string literals are within double quotes.

With C99, we have the introduction of what is known as ‘const’ types, that are constants and we will have more discussions of that in depth when we do C++. So, in C89 the literals are basically fixed values, but in C99, they are considered to be constant type data. So, ‘212’ in C99 it will be considered a const int.

(Refer Slide Time: 09:14)

The slide has a blue header with the title 'Operators'. On the left is a vertical navigation menu for 'Module 01' with topics like Data Types, Variables, Operators, etc. The main content area contains two bullet points:

- An operator denotes a specific operation. C has the following types of operators:
 - Arithmetic Operators: + - * / % ++ --
 - Relational Operators: == != > < >= <=
 - Logical Operators: && || !
 - Bit-wise Operators: & | ^ << >>
 - Assignment Operators: = += -= *= /= ...
 - Miscellaneous Operators: . , sizeof & * ?;
- **Arity of Operators:** Number of operand(s) for an operator
 - +, -, *, & operators can be *unary* (1 operand) or *binary* (2 operands)
 - ==, !=, >, <, >=, <=, &&, ||, +=, -=, *=, /=, &, |, <<, >> can work only as *binary* (2 operands) operators
 - sizeof ! ~ ++ -- can work only as *unary* (1 operand) operators
 - ?: works as *ternary* (3 operands) operator. The condition is operand and the if true logic and if false logic corresponds to other two operands.

At the bottom right is a circular profile picture of the instructor, Partha Pratim Das.

There are several operators in C; you will be familiar with many of them. There are by common or binary operators like addition, subtraction, multiplication. There are unary operations like negation. There are even ternary operations like question mark, colon. Every operator has a fixed arity that is a number of operands that it takes, which could be 1, 2 or 3.

(Refer Slide Time: 09:46)

The slide has a blue header with the title 'Operators'. On the left is a vertical navigation menu for 'Module 01' with topics like Data Types, Variables, Operators, etc. The main content area contains three bullet points:

- **Operator Precedence:** Determines which operator will be performed first in a chain of different operators
The precedence of all the operators mentioned above is in the following order: (left to right – Highest to lowest precedence)
(), [], ++, -, + (unary), -(unary), !", *, &, sizeof, *, /, %, +, -, <, >, ==, !=, |=, *=, /=, &, |, &&, | |, ?:, +=, -=, *=, /=, <<, >>, >=
- **Operator Associativity:** Indicates in what order operators of equal precedence in an expression are applied
- Consider the expression a ^ b ^ c. If the operator ^ has left associativity, this expression would be interpreted as (a ^ b) ^ c. If the operator has right associativity, the expression would be interpreted as a ^ (b ^ c).
 - Right-to-Left: ?:, |=, +=, *=, /=, <<, >>, -, ++, !", *, &, sizeof
 - Left-to-Right: *, /, %, +, -, <<, >>, ==, !=, |=, *=, /=, &, |, &&, | |

At the bottom right is a circular profile picture of the instructor, Partha Pratim Das.

The operator in an expression is evaluated according to their order of precedence. Some operators have higher precedence, some have lower precedence. So, we know that if in the same expression there is multiplication as well as addition; multiplication has to be done earlier wherever it occurs in the expression.

Similarly, if there are more than one of the same operator in an expression then the order of their evaluation will depend on the associativity and some operators are left to right, some operators are right to left. So, here I have shown the different examples. This is just for your reference, you will certainly, if you know this. If you do not, please look up the text to understand this better.

(Refer Slide Time: 10:30)

The slide has a blue header with the title 'Expressions'. On the left, there's a sidebar with a logo and a navigation menu listing topics like Module 01, Partha Pratim Das, Objectives & Outline, History of C, Data Types, Variables, Literals, Operators, Expressions, Statements, Control Flow, Arrays, Functions, Pointers, Expressions, Input / Output, Std Library, Organization, Build Process, References, and Summary. The main content area contains two bullet points under the heading 'Every expression has a value':

- A literal is an expression
- A variable is an expression
- One, two or three expression/s connected by an operator (of appropriate arity) is an expression
- A function call is an expression

Below this, another bullet point says 'Examples:' followed by a code snippet:

- For

```
int i = 10, j = 20, k;
int f(int x, int y) { return x + y; }
```
- Expression are:

```
2.5          // Value = 2.5
i            // Value 10
-i           // Value -10
i = j         // Value -10
k = 6         // Value 6
f(i, j)       // Value 30
i + j == 1 * 3 // Value true
(i == j)? i: 2 // Value 2
```

At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

Now, the next concept in C is an expression. That I have variables and I have operators. I have literals with those I can build expressions. So, expressions are defined in kind of a recursive form will say that every literal is an expression.

If I say number 5, it is an expression by itself. Every variable is an expression and if I have two expressions and connect them by a binary operator then that becomes a new expression. Similarly, I can have expressions with unary operator, ternary operator and so on. Any function call that is done is an expression. So, the basic point is that expression

must have value; anything that has a value in C is called an expression. So, there are different examples you can see here for the variables given here and different expressions are given below.

(Refer Slide Time: 11:34)

Statement

- A statement is a command for a specific action. It has no value
 - A ; (semicolon) is a (null) statement
 - An expression terminated by a ; (semicolon) is a statement
 - A list of one or more statements enclosed within a pair of curly braces { and } or block is a compound statement
 - Control constructs like if, if-else, switch, for, while, do-while, goto, continue, break, return are statements
- Example: *Expression statements*

Expressions	Statements
i + j;	i + j;
k = i + j;	k = i + j;
func(i,j);	func(i,j);
k = func(i,j);	k = func(i,j);

- Example: *Compound statements*

```

{
    int i = 2, j = 3, t;
    t = i;
    i = j;
    j = t;
}
  
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

Now, expressions cannot exist in C by themselves. So, expressions will have to exist as statement. A statement is a smallest unit of command that you can specify in a C program. So, the simplest or the smallest statement that you can have which is called a null statement; is a semicolon itself. Otherwise, if you have an expression, you can terminate that with a semicolon and once you terminate it with a semicolon then it becomes a statement.

So, if you look at the example below, in the expression statement ‘i + j’ is an expression because ‘i’ and ‘j’ are variables and + is an operator connecting them, but the moment you write ‘i + j ;’, it becomes a statement. It can occur independently anywhere, similar examples are shown for assignment for function call and so on.

Besides the expression statement, C has a number of control statements or control constructs, which basically allow the control flow in the program to be managed. So, there are selection statements and loop statements and so on. We will see little bit more

of them in the next slide and if there are number of statements one after the other which need to be grouped for use, and then we put a pair of curly braces around them. We say it becomes a block and such a statement is called a compound statement. So, that whole block of statements is a compound one you can see an example at the bottom.

(Refer Slide Time: 13:19)

The slide has a blue header with the title 'Control Constructs'. On the left, there's a sidebar with a logo and a navigation menu. The main content area contains two columns of bullet points and code examples. The first column lists control statements: Selection-statement (if, if-else, switch), Labeled-statement (Statements labeled with identifier, case, or default), Iteration-statement (for, while, do-while), and Jump-statement (goto, continue, break, return). The second column lists examples of each:

- Selection-statement:**

```
if (a < b) {
    int t;
    t = a;
    a = b;
    b = t;
}
```

```
if (x < 5)
    x = x + 1;
else {
    x = x + 2;
    --y;
}
```
- Labeled-statement:**

```
switch (i) {
    case 1: x = 5;
    break;
    case 3: x = 10;
    default: x = 15;
}
```
- Iteration-statement:**

```
int sum = 0;
for(i = 0; i < 5; ++i) {
    int j = i * i;
    sum += j;
}
while (n) {
    sum *= n;
    if (sum > 20)
        break;
    --n;
}
```
- Jump-statement:**

```
int f(int x) {
    return;
}
```

At the bottom right, there's a video player showing a person speaking.

Now, coming to control constructs which are the key area of a C program which basically tell you, how the execution of the program can happen. We have different ways to control, what will be executed after one statement has been executed. By default we say, the C program has a fall through control, which means that once a statement has been executed then the immediately next statement in the program code will be the next statement to be executed, but we can change that by the control flow.

So, the first kind of control flow is a selection statement; ‘if’ or ‘if else’. So, in the example as you can see that we are saying, if ($a < b$), then if that is true then you do the compound statement that follows it. You can easily understand that what the compound statement is saying that you interchange the value of ‘a’ and ‘b’ by using a third variable.

If you look at the next example of ‘if’, it is showing ‘if else’ kind of statement, where if($x < 5$) that if the condition is true, it does one statement, else if the condition is false

then it does another statement. You can see on that the false part has a compound statement, whereas the true part as a single statement and selection can be done for a multi way. In their multi way form, that you can use the value of a variable.

In this case we have used the variable ‘i’ and you can switch on that depending on what value the variable has taken, you take any one of the cases that are listed. So, if ‘i’ is one then case one will be selected by which ‘x’ will become 6 and we have a default case which is executed, if the value of ‘i’ does not fall among the different cases that exist. Statements like ‘case’ as we have shown in ‘switch’ are also called labeled statement because there is a label to them.

Then we have iteration statements where you can repeat or loop statements very commonly these are called loop statements, where you can have a ‘for’ loop which has three parts. An initial part ‘i’ assigned ‘0’, which is initially done. A second condition part which is checked every time the loop is executed and you continue in the loop provided that condition remains true and there is a body which is basically what follows the ‘for’ statement, which is the sequence of instructions or statements to be executed as a part of the loop and there is an end of loop statement like ‘`++ i`’.

Similarly, we have ‘while’ loop we can have do while iteration and the final type of control statements are ‘go to’, ‘continue’, ‘break’ and ‘return’. As you know, C advises that you should not use ‘go to’. So, we are not showing example of ‘go to’. If you design a C program well then you will not have any reason to use the ‘go to’ at all. So, try to only use ‘continue’ and ‘break’ along with loop and different ‘switch’ statements to achieve your control flow, but you will need ‘return’ to return from a function. So, these are the four types of different types of control constructs that exist.

To sum up, what we have seen in this module so far, we have seen what are the basic components of a C program, which is how do you do a IO, how you; using the data type, how you define variables? How you initialize them? How to form them into expression using operators? How to convert the expressions into statements and different control flow statements to control the flow of the program?

So with this, we will end this part and next we will talk about the derived types and how to use the derived types in C.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 02
Recap of C (Part II)

We have been discussing about C programming in Module 1. This is the part-2, we will now talk about.

(Refer Slide Time: 00:25)

In the earlier part you have seen the basic concepts elementary concepts of C including data types, variables, operators, expressions and statements particularly the control statements. Now, we will move on and we will talk about various derived types in C.

(Refer Slide Time: 00:59)

The slide has a blue header with the title 'Arrays'. On the left, there is a sidebar with a logo and a navigation menu. The main content area contains bullet points and code snippets.

Module 01
Partha Pratim Das
Objectives & Outline
Recap of C
Data Types
Variables
Operators
Expressions
Statements
Control Flow
Arrays
Structures
Unions
Pointers
Functions
Input / Output
Std Library
Organization
Build Process
References
Summary

NPTEL MOOCs Programming in C++

Arrays

- An array is a collection of data items, all of the same type, accessed using a common name

- Declare Arrays:

```
#define SIZE 10
int name[SIZE]; // SIZE must be an integer constant greater than zero
double balance[10];
```
- Initialize Arrays:

```
int primes[5] = {2, 3, 5, 7, 11}; // Size = 5
int primes[] = {2, 3, 5, 7, 11};
int sizeOfPrimes = sizeof(primes)/sizeof(int); // size is 5 by initialization
int primes[5] = {2, 3}; // Size = 5, last 3 elements set to 0
```
- Access Array elements:

```
int primes[5] = {2, 3};
int evenPrime = primes[0]; // Read 1st element
primes[2] = 6; // Write 3rd element
```
- Multidimensional Arrays:

```
int mat[3][4];
for(i = 0; i < 3; ++i)
    for(j = 0; j < 4; ++j)
        mat[i][j] = i * j;
```

Partha Pratim Das 15

As you know the most common derived type in C are arrays. Array is a collection of data items. So, a variable is a single data item, it is a single value; where is array is a collection of one or more data items with the restriction that all data items have to be of the same type. So, if I say there is an array of int, then everything in that all value in that array has to be int. If I say that array is of type char, then every element has to be of type character.

In terms of arrays, the first thing we need to do is to declare an array like we need to declare any variable, so we declare the array in this. So, naturally, we need to specify the name; we need to say what is the type of the element of an array; and also we need to specify what is the maximum number of elements that an array can support.

Now, this maximum number of elements can be specified in multiple different ways. And as we will see that this is one aspect in which arrays in C++ will somewhat differ from arrays in C, but a typical array in C will be defined as double balance within corner brackets 10, which will say that balance is an array of double type elements and maximum 10 such elements could be there. Otherwise, we can define a size by a manifest constant and use that to define declare the array.

Then array can also be initialized, we are moving to the second part. So, arrays can also be initialized that is if I just declare an array then the elements do not have any specified value, but I can declare as well as initialize the array. So, if I say ‘int primes[5]’ initialized with this list of values which are separated by comma and is content within a pair of curly braces then the first value in that list goes to the first element of the array which is the 0th index element as you know.

The second on the list goes to the second element, which is at the index one and so on. Interestingly in C, arrays are allowed to be initialized even without a specifically express size. So, I can just write primes without actually giving a size and just give the initialization list what C compiler does; it finds out how many elements you have initialized it with, and assumes that is the size of the array or those many elements will be there.

So, if primes are just initialized with the list of 2, 3, 5, 7 and 11 there are 5 elements, so it will become an array of five elements. And I have also shown in such cases, since you do not know what is the size, what is the most reliable way to compute the size that is you find what is the total size of the array divided by the size of every element you will certainly get the number of elements that the array contains. It is also possible that you have a bigger array and initialize it with less number of elements; the initialization will always happen from the beginning.

And the remaining elements in such cases will be initialized to 0, but you cannot in it have an initialization list which is bigger than the size of the array that will be an error. An array whether it is initialized or not initialized can be accessed by each and every element. So, next for accessing array elements, we can see that we use the index and we can read or access that element, which exist at that array location. Similarly, I could not also write it on the left hand side of an expression and make an assignment to an array element.

C assumes that every array as it is defined is like a single element also. So, I can define arrays of arrays and of arrays and of arrays and so on; and in this way, I can extend into multiple dimensions. So, these are called multidimensional arrays. So, the simplest of the multidimensional array would be a two-dimensional array, which we commonly in mathematics, we commonly call them as matrix. So, it is given with two dimensions; one is

the number of rows and the number of columns. So, if we define int mat [3][4] that mean, that there are 3 rows and 4 columns in this whole mat array that we have.

So, it makes it two-dimensional; naturally if it is two dimensional, it needs two indices the row index and the column index to be accessed. You can extend it to 3, 4 any higher dimension as it is required, but it is less common that you will have 3 or higher dimensional arrays in regular use.

(Refer Slide Time: 06:35)

Structures

- A structure is a collection of data items of different types. Data items are called *members*. The size of a structure is the sum of the size of its members.

- Declare Structures:

```
struct Complex { // Complex Number
    double re; // Real component
    double im; // Imaginary component
} c; // c is a variable of struct Complex type
printf("size = %d\n", sizeof(struct Complex)); // Prints: size = 16
```

- typedef struct _Books {
 char title[50]; // data member
 char author[50]; // data member
 int book_id; // data member
} Books; // Books is an alias for struct _Books type

- Initialize Structures:

```
struct Complex x = {2.0, 3.5}; // Both members
struct Complex y = {4,3}; // Only the first member
```

- Access Structure members:

```
struct Complex z = {2.0, 3.5};
double norm = sqrt(z.re*z.re + z.im*z.im); // Using . (dot) operator
```

```
Books book;
book.book_id = 6495407;
strcpy(book.title, "C Programming");
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 16

Next to arrays are structures. So, like array is a collection and we will slowly start using the term container a term which is not so common in C, but we will see in C++ and particularly the standard library that comes with C++, the terms commonly used is called container because container is something which can contain other elements. Array is a container; it is a collection of data items with the specific behavior that all items are of the same type as we have seen.

Now, structure in contrast is also a container; it is a collection of data items, but here the data items could be of different types; it is not necessary that they have to be of different types, but they can be of different types. So, if I look into a structure, these data items are often called

members; and we will more and more use this term member or data member to mean components or data items of a structure, we show a simple example of forming a complex number.

As we know complex numbers have two components - the real part and the imaginary part. So, each part can be a double. So, we define that with struct complex that tells us that there are two components; in this case, both of them are of the same type and then C is declared to be a variable of this structure type. We can also, in the next one; we show that you truly have one structure where components are of different type. So, the first two components in the books structure are title and author; they themselves are arrays of character which is basically what it means that they will become C strings of title name and the author name; and the third component is actually an integer, which is keeping the book id.

Structures could be defined directly as by their name, and using the keyword struct we can use that or the struct and the structure name can together be given and alias by the use of type def keyword. Type def is a short form of type definition. And as we go into C++, we will talk more about this that why type def actually is not a type definition, it is kind of a misnomer, but historically it has this keyword has been used, and it continue to be used in C, but it is basically type alias. It is just another name given to the struct complex of the struct books, as we have. It just becomes it easier.

If you use, have a type def then you can directly use it. So, if we come to the access of the structures then we can see that since complex was just defined as struct complex to use that we need to write struct complex and then the variable name followed by the initialization of course. But books here was defined by type def, so books does not need to be written with struct books I can just write books, it has given a total name for the struct book. So, it becomes easier to use it in that way. Now, any structure variable can be initialized like a simple variable; and the notation is very similar to the way we initialized arrays.

In contrast, in arrays, the list formed within curly braces meant different components or different indexed elements of the array here; in case of structure, it means the different components or the data members of the array; and the way you list these initialization values, at the way the data members are listed from the top to bottom order. So, if we look into the

specific cases given in this code, then you have we are showing struct complex x initialized with 2.0, 3.5, which means 2.0 will be the initial value of the first component which is ‘re’. So, you just read them from top to bottom and read this list from left to right and correspond to them. So, 3.5 will be the initial value of ‘im’. So, x will get become a complex number $2.0 + j 3.5$, if we think of it that way.

It is possible that we initialize only one or some of the members, but you can do that only from the initial part starting from the top. You cannot skip ‘re’ and just initialize ‘im’, but you can initialize ‘re’ as you are doing here with 4.2 and skip ‘im’, which is not mentioned here. Then every component of a structure can be accessed by the dot notation, the dot operator we are showing here. So, if x is a complex variable, a structure variable, and then the ‘re’ component of this variable x is written as x.re. Similarly here book is a structure of the above type books type, so it has an ‘id’ component which is book.id. So, we use the dot notation to access the structure component.

So, we can see among the two most commonly used, two most powerful containers in C have two different ways of access; array since all elements are of uniform type. It is accessed by position, it is called the positional access, because you want to find out in the array starting from the beginning the zeroth, first, second, so you just go by number, so array is also called a indexed container, because you can access the elements by number.

In contrast, in structure, the elements are of different components are of different types. So, there is not much significance to what is a first component in the list or what is a third component in the list and so on. So, you access the elements by name. Now this is also a style, which we will see in different languages as to whether you can access something by name or you can access something by position, and similar issues of access will come up when we talk about function arguments also which in C as you know is also by position.

And I would just like to I would like that these are few common aspects of programming languages that you need to be careful about as to when you have a list of elements what is the access mechanism between whether it is positional or it is by name. So, in structure, we see one kind; in array, we see the other.

(Refer Slide Time: 13:54)

The slide is titled "Unions" and is part of Module 01 by Partha Pratim Das. It covers the following topics:

- A union is a special structure that allocates memory only for the largest data member and holds only one member at a time.
- Declare Union:

```
typedef union _Packet { // Mixed Data Packet
    int iData;           // integer data
    double dData;        // floating point data
    char cData;          // character data
} Packet;
```

printf("size = %d\n", sizeof(Packet)); // Prints: size = 8
- Initialize Union:

```
Packet p = {10}; // Initialize only with a value of the type of first member
printf("iData = %d\n", p.iData); // Prints: iData = 10
```
- Access Union members:

```
p.iData = 2;
printf("iData = %d\n", p.iData); // Prints: iData = 2
p.dData = 2.2;
printf("dData = %f\n", p.dData); // Prints: dData = 2.200000
p.cData = 'a';
printf("cData = %c\n", p.cData); // Prints: cData = a

p.iData = 97;
printf("iData = %d\n", p.iData); // Prints: iData = 97
printf("dData = %f\n", p.dData); // Prints: dData = 2.199999
printf("cData = %c\n", p.cData); // Prints: cData = a
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 17

Now moving on a C supports another kind of collection container, which is pretty much like structures, they are called union. Now, the only difference is instead of using the struct keyword you use the union keyword. The way you specify or way you access the members are same between structure and union. But what is different is the way the memory is allocated. If you have a structure with two, three different members then when a variable of that structure type is created all members are allocated area. So, if you go back to this structure, then you will see that here after the complex structure has been defined it has two components. So, both these components are actually allocated.

So, if we assume that every double texts bytes then the size of the whole complex structure is 8 bytes plus 8 bytes that is 16 bytes, because both exist. In contrast, in union, it is the allocation is done in memory for only the largest component. So, you make sure that your basic assumption is as name union suggests that this is a union of all these fields, so at any point of time only of the components will exist only one of the components has to be assumed. So, naturally if you are representing only one of them you need enough space for the largest one that is only the logical thing. So, if we look into this union packet then we will see that in packet, 3 components are of int double and char type.

So, if we take a reasonable assumption about the size in a say 32-bit machine then int will be 4 bytes, double will be 8 bytes and char possibly will be 2 bytes. So, the size of packet will be 8 bytes which is the size of the largest component double. So, the interesting aspect is this on one side allows you to minimize the size of the whole record, the size of the whole container, if you know that you do not need all the components at the same time.

So the flip side is when you initialize you were initializing only the value of the first component because certainly here since you can have only one component, we have space for only one component the initialization cannot have a list of components list of constants to initialize different data members. And we have seen that in initialization can be partial provided I have initialized the first one or the first two or the first three here since there is only one data member there is only one value that you can initialize and that has to be the type of the first value.

Otherwise, you will have to you will not need to initialize rather you will simply take the component and assign that. So, in terms of access, the same dot notation the dot operator will be useful for accessing the different components, but the point is you will have to remember that only one value is maintained out of though we have three components there is only one value.

So, depending on what is the component that you have assigned last when you access the value of that component only will be valid. If you access a different component then what was assigned last then you will get unexpected result. So, in this access code if you look at little carefully, then you will see in the first part of the code we saw that iData the integer we assigned to and then we access iData, so 2 is printed. In double data, we do 2.5 access double data 2.2, so double data we access the 2.2 is printed. In the char data, we assign constant character ‘a’, we access that ‘a’ is printed. So, we whatever you access last are what are printed.

Now, in the next, we show an interesting thing we in the iData there is a integer part we assigned 97, and then we accessed the integer data; obviously, 97 is printed the last. But if without assigning anything we access it as dData then you get some 2.1999 which is not something very meaningful. So, this is happening because the dData, double data has much

larger size 8 bytes; whereas you had assigned iData which means that you have only assigned 4 bytes of that; remaining 4 bytes have some old garbage value. So, what you get as dData is completely wrong.

The final interesting thing is if you access it as C data then you are actually accessing 97 as C data, and you see you are getting ‘a’ why, 97 is the ASCII code of ‘a’. Now the interesting thing is that C data possibly is 1 byte here, which means that it can contain up to 255 the value incidentally that you have given to the 4 bytes of int happens to be less than 255. So, we can easily understand that in this int the higher three bytes are all zeros. So, when I access it as a character I just get the 97 that I have stored there. So, it feels as if I have got the corrected value which is the code of a. The point that I am trying to highlight is it is one value that resides, because there is only one memory location; you will have to be very careful about this when you are using union.

The reason union was given in C is for the fact that if you have to make an aggregate, if you have to make a container which can deal with data of variety of types, typically just to give you the kind of reference where union is used is if you think of a network connection, and you have a network port, where variety of data is coming different kinds of data packets are coming, and you do not know what kind of data packet will come out of a possible say 10 different kinds of data packets.

But you know that at any point of time only one kind of data packet will come, only one data packet will come then how do you define a programming structure to store this data packet. Now, you do not want to make a structure which has the possibility of storing all these 10 different kinds of data packets, because that will unnecessarily take a lot of space. But at the same time, utilizing the fact that only one type of packet will come at 1.0 of time you could use this union structure as a collection of also that is how it has been used in C.

And we will see in terms going to C++ there will be a lot of ramifications of this concept of union C++ does provide us with a very strong feature based on object orientation something known as inheritance and specialization. And we will see how similar effects can be created in C++ without using union, so that is just for your comments, we will come to with the details subsequently when we go to that level of C++.

(Refer Slide Time: 21:55)

The slide has a blue header with the title 'Pointers'. On the left, there's a vertical sidebar with a logo at the top and a list of navigation links: Module 01, Partha Pratim Das, Objectives & Outline, Review of C, Data Types, Variables, Data in Functions, Expressions, Statements, Control Flow, Arrays, Structures, Unions, Pointers, Functions, Input / Output, Standard Library, Debugging, Build Process, References, Summary. Below the sidebar, the main content area starts with two bullet points: 'A pointer is a variable whose value is a memory address' and 'The type of a pointer is determined by the type of its pointee'. It then shows a code snippet:

```
int *ip; // pointer to an integer
double *dp; // pointer to a double
float *fp; // pointer to a float
char *ch; // pointer to a character
```

Following this, another bullet point says 'Using a pointer:' and shows a C program:

```
int main() {
    int i = 20; // variable declaration
    int *ip; // pointer declaration
    ip = &i; // store address of i in pointer

    printf("Address of variable: %p\n", &i); // Prints: Address of variable : 00ABF79C
    printf("Value of pointer: %p\n", ip); // Prints: Value of pointer : 00ABF79C
    printf("Value of pointee: %d\n", *ip); // Prints: Value of pointee : 20
    return 0;
}
```

At the bottom of the slide, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

Now next the most interesting and possibly one of the most powerful feature of C programming that comes up. C programming, if you are familiar with little bit of the history of the language, and I seek this to be an appropriate point to introduce that to you, it is before C was done in an university by a group of computer scientists, professors typically, and a group of professionals, we were trying to write an operating system which later on became popular as Unix, you all use Linux which is a much later generation of that. So, while they were trying to write Unix, trying to write code for Unix, they needed a programming language, because otherwise how do you write code.

Now prior to Unix, there was no high level programming language of the type that you see in C or C++ or java or python available, where you could write an operating system, because when you want to write an operating system, you need to not only deal with values, but also you need to deal with memory. Because the programs will finally, get stored in memory that where data will reside in memory. So, when we write a program, we are just dealing with variables, we are just interested about values. We can write a complete program like in python or in java without at all thinking of where these values are stored in memory, but you cannot write an operating system assuming that.

You need to that programming language to be aware of the memory or to be aware of the address where values get stored. So that was one of the various reasons for which the team of Kerning, Ritchie, Morris buck and all others, we were in the Unix team needed to do a programming language, and they quickly designed C which then later on became a default language for all of us, rest of it as they say is history but this is the genesis for which C for the first time introduced the strong concept of managing addresses as data and that is what we say is a pointer.

I just wanted to give you the reason why you needed this, but when you have this feature of dealing with address as data that gives C a very strong background; in which C can create variety of data structures as you all some of you all have already done. Like you cannot think of doing a list without having pointers; it is possible, it is possible you can have create a link list by using two arrays. An array to keep the index the address of where you will find the next element; and another array to actually have the values, but that is not something, which is efficient which is scalable which what people would do; so, you will always use pointers.

So, pointers are the next derived types as you all know. So, it is a variable whose value is a memory address. And the type of a pointer is decided by not by the pointer itself, all pointers are addresses. So, address which has only one type that is of the pointed type, but their type is decided based on what kind of value they are actually pointing to. So, if I have an int *ip then it is pointing to an integer type of value.

So, we will say it is int * type. Now, to use pointer, you will be familiar with this then I can have a variable i which is initialize to 20, I have a pointer ip which I would use to point to i, which is int *ip and there is a special operator ampersand as you know which I can use with any variable to get the address of the memory location, where this variable will be stored. So, I can take that address and store that address in the pointer variable ip.

And once I have done that then if I try to print the address of variable i or i print the address or the value of ip, which is actually the address that I have stored here then they will certainly be identical that is what we are showing here. Now given that address, I can actually find out what value exists at that pointed location by using the star or content of operator. So, here along with the pointer, we will always use the address of as well as content of operators.

(Refer Slide Time: 26:49)

The slide has a blue header with the title 'Pointers'. On the left, there's a sidebar with navigation links for 'Module 01' (Partha Pratim Das), 'Objectives & Outline', 'Topics of C', 'Data Types', 'Variables', 'Operators', 'Decision', 'Loops', 'Pointers', 'Functions', 'Memory / Dynamic', 'Std Library', 'Declarations', 'Build Process', 'References', and 'Summary'. The main content area contains two sections: 'Pointer-Array Duality' and 'Pointer to a structure'. Below these are examples of code. A video player window on the right shows Partha Pratim Das speaking.

```
• Pointer-Array Duality
int a[] = {1, 2, 3, 4, 5};
int *p;
p = a;
printf("a[0] = %d\n", *p); // a[0] = 1
printf("a[1] = %d\n", *(p+1)); // a[1] = 2
printf("a[2] = %d\n", *(p+2)); // a[2] = 3
p = &a[2];
*p = -10;
printf("a[2] = %d\n", a[2]); // a[2] = -10

• malloc-free
int *p = (int *)malloc(sizeof(int));
*p = 0x8F7E1A2B;
printf("%X\n", *p); // 8F7E1A2B

unsigned char *q = p;
printf("%X\n", *q++); // 2B
printf("%X\n", *q++); // 1A
printf("%X\n", *q++); // 7E
printf("%X\n", *q++); // 0F

free(p);

• Pointer to a structure
struct Complex { // Complex Number
    double re; // Real component
    double im; // Imaginary component
} c = { 0.0, 0.0 };

struct Complex *p = &c;
(*p).re = 2.5;
p->im = 3.6;
printf("re = %lf\n", c.re); // re = 2.500000
printf("im = %lf\n", c.im); // im = 3.600000
```

Now pointers can be used in multiple different ways; they had very powerful in terms of creating various idioms in the programming language. The first and the most common is the duality between the pointer and an array; and most easily seen in terms of a one dimensional array. So, array is a series of locations; and pointer is address of the starting location of the array. So, I can take the array and assign it to a pointer variable. And then if I just do `*p`, it gives me the content which will be the content at the starting location of the array which will happen to be a 0.

I can increment a pointer and that is a very interesting concept, if I increment a pointer then the amount by which it is incremented is not 1, the amount by which it is incremented is the size of the type of element it is pointing to type of value it is pointing to. So, if it is pointing to an int and the `sizeof(int)` in the system is 4, then the pointer value actually will increment by 4, so that in terms of the array now you are pointing to the second location. So, you can see that if I do `* + + p`, it will first increment.

So, it is now pointing to element 1 and then it takes the element 1, which is basically 2. Similarly, I can have I can take `p + 1` as an expression, `p + 1`; similarly is the current location of the pointer plus 1 element size what that can be. Pointers can be used with structures, if we

use that then we can access the elements by `*p`, and then `*p` is a structure that it is pointing to dot re is a component of that structure.

This can be shortened by a de referencing operator as this given in C, you know all this. So, we will skip that then pointers can also used in terms of dynamic allocation. So, I can dynamically allocate using malloc, and I get a pointer which does not have a specified type we could say that it is a void star. And we can use a casting which is forcibly to be done by the programmer to cast that to integer type value. This is an interesting code given here, I will not go through the details; try to understand how this code works how to manipulate with pointers. And if you have questions, you can ask us on the blog. And we can use pointers for dynamically allocating arrays as well.

With this, in this part of the recap, we have primarily talked of the different derived types. First, we have talked about the containers, arrays, structures and unions; the three main types of containers that C provide; and we have talked of managing the different variables and addresses through pointers. In the next, we will talk about the functions.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 03
Recap of C (Part III)

We will continue on Module 01, recapitalization of C. This is the third part. In the first two parts we have talked about the data types, variables, expressions, statements. And, in the second part we have talked about different derived types, arrays, structure, union and pointer.

In the spot, we will start with the basic modular concept of C, which is a function.

(Refer Slide Time: 00:51)

The slide has a dark blue header with the title 'Functions'. On the left is a vertical navigation menu with the following items:

- Module 01
- Partha Pratim Das
- Objectives & Outline
- Recap of C
 - Data Types
 - Variables
 - Literals
 - Operators
 - Expressions
 - Statements
 - Control Flow
 - Arrays
 - Structures
 - Unions
 - Pointers
 - Functions** (highlighted)
 - Input / Output
- Std Library
- Organization
- Build Process
- References
- Summary

The main content area contains the following text and code snippets:

- A function performs a specific task or computation
 - Has 0, 1, or more parameters / arguments. Every argument has a type (void for no argument)
 - May or may not return a result. Return value has a type (void for no result)
 - Function declaration:

```
// Function Prototype / Header / Signature
// Name of the function: funct
// Parameters: x and y. Types of parameters: int
// Return type: int

int funct(int x, int y);
```
 - Function definition:

```
// Function Implementation
int funct(int x, int y)

// Function Body
{
    return (x + y);
}
```

So, as you all know a function performs a specific task of computation. A function is like can be treated as a black box, which can take a number of parameters and will give you a result. Now, while usually we will expect a function to take one or more parameters, it is possible that you write a function which does not take any parameter. And, it is typical that it gives you a value at the end of computation; we say it returns a result. But, it is also possible that it may not return any result.

The number of parameters that a function has; each parameter also called argument, we will have a type. And, if we do not want to specify the type we can use void. Return will also have a type. And, we will use void if there is nothing to return.

A typical declaration will look like this; funct is the name of the function. On the left, a return type and on the right within this pair of parenthesis, we have the parameters. If there are more than one parameter, they are separated by comma. And, at the end of this parenthesis if you put a semicolon, then we know that you are just interested to talk about what parameters the function takes and what type of value it returns. But, you are not interested to specify how actually this funct function computes the result integer from the parameters x and y.

In such cases if you just dominate the list of arguments with the semicolon, we will say this is a function header or a function prototype or a function signature. And more and more, the signature or prototype kind of terms will keep on occurring in C++. And since in this case, in case of a signature we are not actually specifying how x and y will be used to compute the result. It is optional whether you specify x or y or both of them. You could just write it as int funct (int , int); that will also be a valid header.

So, what this tells us? It tells us that there are two parameters. First parameter is an integer; second parameter is another integer. It tells that the name of the function is funct, it tells us that the type of value it will return is int. This is the purpose of the function declaration or the function header.

Now, when we are ready to specify as to what this function will compute or how this function will compute the result from the parameters, then we provide the function body; which is the whole function body is a compound statement. So, it is a pair of parenthesis, curly braces, within which the function body has to be return. So, within that there could be multiple declarations and statements specifying the function body.

A function will have a return statement, which returns an expression of the return type as a final result. If the function is not returning anything, if the function return type is void, then the return statement will not have an expression associated with it. Please note that

in C 89, it was allowed that if a function does not return anything, then it is not necessary to specify the return statement. That protocol still continues. But, for several reasons that will become clear, when we do more of C++. It is strictly avoidable that you write a function and do not put a return. So, even though you may not return anything from the function that is return type is void, please provide a return statement.

(Refer Slide Time: 05:04)

Module 01
Partha Pratim Das

Objectives & Outline

Recap of C

- Data Types
- Variables
- Literals
- Operators
- Expressions
- Statements
- Control Flow
- Arrays
- Structures
- Unions
- Pointers
- Functions**
- Input / Output

Std Library

Organization

Build Process

References

Summary

Functions

- **Call-by-value** mechanism for passing arguments. The value of an actual parameter copied to the formal parameter
- **Return-by-value** mechanism to return the value, if any.

```

int funct(int x, int y) {
    ++x; ++y;           // Formal parameters changed
    return (x + y);
}

int main() {
    int a = 5, b = 10, z;

    printf("a = %d, b = %d\n", a, b); // prints: a = 5, b = 10
    z = funct(a, b); // function call by value
    // a copied to x. x becomes 5
    // b copied to y. y becomes 10
    // x in funct changes to 6 (++x)
    // y in funct changes to 11 (++y)
    // return value (x + y) copied to z

    printf("funct = %d\n", z); // prints: funct = 17
    // Actual parameters do not change on return (call-by-value)
    printf("a = %d, b = %d\n", a, b); // prints: a = 5, b = 10

    return 0;
}

```

NPTEL MOOCs Programming in C++

Partha Pratim Das

Now, functions get their parameters by a mechanism, which is known as call by value. I assume that you all know this where at the call site, the example is here. On top, you can see the function body, the whole definition of the function. And, here you see the function invocation or function call, where we are using two parameters that are local to. That their variables local to main to call this function. So the result of this, the first parameter ‘a’ is copied to the first formal parameter x. So, ‘a’ ‘b’ are called actual parameters at the call site; ‘x’, ‘y’ are called the formal parameters at the definition site.

And, as I had mentioned in reference to accessing different components of an array and accessing different components of a structure that there are different conventions. Here C follows a positional parameter convention to call function.

So, the first actual parameter corresponds to the first formal parameter; the second actual

parameter corresponds to the second actual formal parameter and so on. So, you do not care about the name of the formal parameter at all. The actual parameter value is copied from one parameter to the next and then it is used in the function. Since these are copies, so they have separate memory locations. So, after when funct() starts executing, x will have a location different from ‘a’.

But, since the value has been copied, the value of ‘a’ is 5. Value of x also to start with will be 5. But, then within the funct body, we have made changes to x and y. We have incremented x, incremented y. We have computed the return value. So, when the function returns, that is, when this value, return value $x+y$ is computed and given back to the caller main(), and that value is copied to z. When that happens, then we lose the values of the formal parameters x and y. interestingly, a and b which are actual parameters, they will not change, even though x and y had changed.

And, it is very easy to see. In call by value, you are using separate location. You just, at the beginning you had copied the values. So, since you had copied the values at the beginning, the different locations, when you lose x and y, all the changes we have made to x or to y or to both, only simply gets lost. The actual parameters are not affected.

So, just to illustrate that, we are printing the values of ‘a’ and ‘b’ here. And, we find that they are as original. They are still 5 and 10, even though x and y, the copied values had changed. Which is not very common particularly to the C programmers. Or, it is a notion that the mechanism by which you return the value is called a return by value mechanism.

And, it is not explicitly mentioned in C because there is no other way to return, to put it straight. I mean, see the only way to return is by copying the value back; because at this point, the function ‘funct’ which was executing is terminated; will get terminated after giving you the value. So, there is no other way than to keep a copy of that value, which we have assigned to z here, to get that value back. As you go to C++, we will see a different mechanism both for call by value as well as for return by value, which will be very powerful.

(Refer Slide Time: 08:53)

The slide is titled "Functions" and is part of "Module 01" by Partha Pratim Das. The sidebar contains a navigation menu with topics like Data Types, Variables, Literals, Operators, Expressions, Statements, Control Flow, Arrays, Structures, Unions, Pointers, Functions, Input / Output, Std Library, Organization, Build Process, References, and Summary. The main content area discusses recursive functions, noting they have a recursive step and an exit condition. It includes two code snippets: one for factorial and one for calculating the number of 1's in the binary representation of a number. A video thumbnail of Partha Pratim Das is on the right.

- A function may be recursive (call itself)
 - Has recursive step/s
 - Has exit condition/s
- Example:

```
// Factorial of n
unsigned int factorial(unsigned int n) {
    if (n > 0)
        return n * factorial(n - 1); // Recursive step
    else
        return 1; // Exit condition
}

// Number of 1's in the binary representation of n
unsigned int nOnes(unsigned int n) {
    if (n == 0)
        return 0; // Exit condition
    else // Recursive steps
        if (n % 2 == 0)
            return nOnes(n / 2);
        else
            return nOnes(n / 2) + 1;
}
```

Functions can be recursive. I am sure you all have seen this. You all have seen factorial functions like this or the very famous Fibonacci function, which uses two recursive calls. We have seen merge sort, we have seen quick sort; these are all very typical examples of recursive function. Any recursive functions will have a recursive step and one or more exit condition to end the recursion.

Here is another example, which is less commonly used. So, I just put it here for illustration. You have given an unsigned integer n. This recursive function computes a number of 1's in the binary representation of n. You just go through this carefully and is familiarly with itself with the recursion mechanism.

(Refer Slide Time: 09:38)



Function pointers

Module 01
Partha Pratim Das

Objectives & Outline
Recap of C
Data Types
Variables
Literals
Operators
Expressions
Statements
Control Flow
Arrays
Structures
Unions
Pointers
Functions
Input / Output
Std Library
Organization
Build Process
References
Summary

```
#include <stdio.h>
struct GeoObject {
    enum { CIR = 0, REC, TRG } gCode;
    union {
        struct Cir { double x, y, r; } c;
        struct Rec { double x, y, w, h; } r;
        struct Trg { double x, y, b, h; } t;
    };
};

typedef void(*DrawFunc) (struct GeoObject);

void drawCir(struct GeoObject go) {
    printf("Circle: (%lf, %lf, %lf)\n",
        go.c.x, go.c.y, go.c.r);
}

void drawRec(struct GeoObject go) {
    printf("Rect: (%lf, %lf, %lf, %lf)\n",
        go.r.x, go.r.y, go.r.w, go.r.h);
}

void drawTrg(struct GeoObject go) {
    printf("Triag: (%lf, %lf, %lf, %lf)\n",
        go.t.x, go.t.y, go.t.b, go.t.h);
}

DrawFunc DrawArr[] = { // Array of func. ptrs
    drawCir, drawRec, drawTrg
};

int main() {
    struct GeoObject go;
    go.gCode = CIR;
    go.c.x = 2.3; go.c.y = 3.6;
    go.c.r = 1.2;
    DrawArr[go.gCode](go); // Call by ptr

    go.gCode = REC;
    go.r.x = 4.5; go.r.y = 1.9;
    go.r.w = 4.2; go.r.h = 3.8;
    DrawArr[go.gCode](go); // Call by ptr

    go.gCode = TRG;
    go.t.x = 3.1; go.t.y = 2.8;
    go.t.b = 4.4; go.t.h = 2.7;
    DrawArr[go.gCode](go); // Call by ptr
    return 0;
}
```

Circle: (2.300000, 3.600000, 1.200000)
 Rect: (4.500000, 1.900000, 4.200000, 3.800000)
 Triag: (3.100000, 2.800000, 4.400000, 2.700000)

Partha Pratim Das

Functions and pointers can be mixed in a very interesting way. And, this is again one of the very powerful features of C. Before highlighting what we want to specify for the C languages as such, let us just briefly understand this example; because this is an explanation by example.

So, what we are showing at the top, the situation is like this; that we have a collection of geometric objects. The geometric objects could be circle, rectangle or triangles. We assume that the triangles to be right angled triangles, which are aligned with the axis. So, we have structure for each of one them. So, a circle is x, y that the center and the radius, rectangle is x, y, that is, say left bottom corner with a height and a triangle is x, y, which is the right angled corner, the base and the height. And, here we make a structure. Again, this is a typical way to write in C. We make a union of all these because a geometric object is any one of them. And given a particular GeoObject, it could be either a circle or a rectangle or a triangle.

Now, how do I know which one is done? So, for that in this structure we are using another enumerated value, gCode, which keeps one of the codes of Cir, Rec or Trg. So, the idea is if you have put, in this union if you have put the structure for a circle you set gCode to cir, which is 0; if you have put in this union, the structure for a rectangle, then

you put gCode as Rec and so on. So, this is a typical use. So, you can see that we have a union of structures.

And then, we have a structure containing that union and indexing code to understand what that union has. And, if you, in C programming this was a very common paradigm to do. And, we will use this example later on in C++, as I said in the context of inheritance. And, show that how in C++ this can be done lot more efficiently.

Now, the situation is since these are three different types of geometric objects and my task is to draw them on the screen. So I have, now the way you draw a circle and the way you draw a rectangle and way you draw a right angled triangle are all different. So, naturally one pro code cannot be written. One function cannot be written, which can draw all three of them. So, you assume that there are 3 different functions

In C, all of them are actually drawing. But in C, certainly we cannot have two functions having the same name. So, I call them as DrawCircle, draw a rectangle, draw triangle. Now, each one of them takes the same structure GeoObject. For example, if I look into DrawCircle, then it takes the GeoObject. And, in the GeoObject it will be able to find out that it indeed is a circle, if it checks for the gCode. And, it will print all these values. Here Actually this, in the real DrawFuncion, the print will not be there. There will be graphics function calls to actually draw the circle. But just to illustrate the point, I am showing you here through prints. So, these are the three different functions, which can draw three different types of geometric objects that exist.

Now, we want to write a routine main function, which given the gCode of any of this objects. And, the structural parameters of that object should be able to draw it. So, the idea is, I would like to call the DrawFuncion. Any of this DrawFuncions not using their name. But, from the fact that if my gCode is Cir and I want to call the draw, this draw should get called. But, if my gCode is Rec and I want to call this, drawRec should get called and so on.

So, we want to make the call uniform, so that I do not really need to know what particular object I have at this point of time. What object I have is already a part of the

object structure; because I have a gCode. And then, it should be possible that the corresponding function will get called. The mechanism that is done here is I create an array called DrawFuncion and put all this function names. As you put this function names, these are called function pointers. A function name used by itself without the pair of parenthesis; the pair of parentheses, as we will see is called function operators, which tells you that the function is being invoked.

Here those operators are not used. So, these are just the function names. They are the function pointers. Or, in other words these are the address where the respective function starts. So, this is an array. DrawArrr is an array of these function pointers. And, what will you see? That the zeroth entry in this is DrawCir, which is drawing the circle. So, if the gCode is zero, that is, Cir. And then, if I access this array with that gCode, so go dot gCode is Cir here, which is zero; which means the zeroth function pointers.

So, the value of DrawArrr go dot gCode turns out to be DrawCir, whereas if I put go dot gCode as Rec, if it is a rectangle structure, then go dot gCode here is 1. After zero, this is 1; which means in the array I am accessing location 1. So, this particular function becomes drawRec function and so on.

So, the notation is little bit not so common. So, you do not see the typical function like name. But you, basically it looks like an array element which is the pointer. And, but when you get the pair of parentheses, you know that there is a function operator which invokes the function that you have access from this array.

Now, to be able to create a function pointer or array of function pointers, what does an array need? Array needs all elements must be of the same time. So, all these function pointers must be of the same type; which means they must take the same type of parameters and must return the same type of value. So, we ensure that by creating a type or aliasing a type DrawFunc here by type def; which says that it takes a structure GeoObject and returns a void. We want to say that this is a pointer type. That is why the name is not DrawFunc. The name is given as star DrawFunc. The way, if we have to define an integer pointer we say int *p. So, basically the name is not p. Name is as if you are saying that the *p is an integer. So, here what you are saying is ‘*DrawFunc’ is a

function which takes a GeoObject, gives a void.

And, now if you look into all these three candidate functions, all of them have that same signature. So, they have the uniform signature. So, they are all of this DrawFunc type, which is a pointer to a function taking a structure geo, struct GeoObject and returning nothing. So, this is a typical use of; the function pointers have very powerful mechanism in C. If you; all the graphic systems, the menus, everything you use is this concept of function pointers. And, as we go to C++, we will show that is how C++ is significantly improved this and made it easier and more robust to use, in terms of what is known as a virtual function table. That will come when we discuss that.

(Refer Slide Time: 18:16)

The slide has a dark blue header with the title "Input / Output". On the left, there's a sidebar with the NPTEL logo and a vertical list of topics under "Module 01" by Partha Pratim Das. The topics include: Objectives & Outline, Recap of C, Data Types, Variables, Literals, Operators, Expressions, Statements, Control Flow, Arrays, Structures, Unions, Pointers, Functions, Input / Output (which is highlighted in yellow), Std Library, Organization, Build Process, References, and Summary. The main content area contains a bulleted list of three points about printf and scanf, followed by a C code example:

- int printf(const char *format, ...) writes to stdout by the format and returns the number of characters written
- int scanf(const char *format, ...) reads from stdin by the format and returns the number of characters read
- Use %s, %d, %c, %lf, to print/scan string, int, char, double

```
#include <stdio.h>
int main() {
    char str[100];
    int i;
    printf("Enter a value :");           // prints a constant string
    scanf("%s %d", str, &i);           // scans a string value and an integer value
    printf("You entered: %s %d ", str, i); // prints string and integer
    return 0;
}
```

At the bottom, the footer says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" with slide number "24".

Finally, to close the course on C programming language, you know there are input, output functions available in the stdio.h. These are common; most common are printf and scanf, which use format strings. It uses; also another interesting part of C programming known as ellipsis or these are known as variadic functions, given that you do not know how many parameters you will put in printf.

So, we ever wrote the code of printf, did not know whether you want to print 5 values or 10 values or 1 value. So, a typical format bit structure has been created. I will not go into

depth of that. We will cross that when we come to dealing with discussing I/O s in C++. To contrast, how C++ avoids variadic input output functions. But, these are the typical ways to do input output in C, which I am sure all of you are very familiar with.

(Refer Slide Time: 19:13)

Module 01
Partha Pratim Das

Objectives & Outline

Recap of C
Data Types
Variables
Literals
Operators
Expressions
Statements
Control Flow
Arrays
Structures
Unions
Pointers
Functions
Input / Output

Std Library
Organization
Build Process
References
Summary

NPTEL MOOCs Programming in C++

Partha Pratim Das

To write to or read from file:

```
#include <stdio.h>
int main() {
    FILE *fp = NULL;
    int i;
    fp = fopen("Input.dat", "r");
    fscanf(fp, "%d", &i);
    fclose(fp);

    fp = fopen("Output.dat", "w");
    fprintf("%d\n", i*i);
    fclose(fp);
    return 0;
}
```

These are examples showing you how to do the input output with array, with files. So, use a file pointer, you do f open to open that and so on.

(Refer Slide Time: 19:25)

The slide has a blue header bar with the title "C Standard Library". Below the header is a sidebar on the left containing the following navigation links:

- Module 01
- Partha Pratim Das
- Objectives & Outline
- Recap of C
- Data Types
- Variables
- Literals
- Operators
- Expressions
- Statements
- Control Flow
- Arrays
- Structures
- Unions
- Pointers
- Functions
- Input / Output
- Std Library
- Organization
- Build Process
- References
- Summary

The main content area starts with a section titled "Common Library Components:" followed by a table:

Component	Data Types, Manifest Constants, Macros, Functions, ...
stdio.h	Formatted and un-formatted file input and output including functions <ul style="list-style-type: none">• printf, scanf, fprintf, fscanf, sprintf, sscanf, feof, etc.
stdlib.h	Memory allocation, process control, conversions, pseudo-random numbers, searching, sorting <ul style="list-style-type: none">• malloc, free, exit, abort, atoi, strtold, rand, bsearch, qsort, etc.
string.h	Manipulation of C strings and arrays <ul style="list-style-type: none">• strcat, strcpy, strcmp, strlen, strtok, memcpy, memmove, etc.
math.h	Common mathematical operations and transformations <ul style="list-style-type: none">• cos, sin, tan, acos, asin, atan, exp, log, pow, sqrt
errno.h	Macros for reporting and retrieving error conditions from error codes stored in a static memory location called errno <ul style="list-style-type: none">• EDOM (parameter outside a function's domain – so on)• ERANGE (result outside a function's range), or• EILSEQ (an illegal byte sequence), etc.

At the bottom of the slide, there is a footer bar with the text "NPTEL MOOCs Programming in C++" and "Partha Pratim Das". To the right of the footer, there is a circular profile picture of Partha Pratim Das.

Now, we will move on to. Just mention that if you just had the C language, then you could not have written any program because you saw the first program which we discussed in the first part of this module; “Hello World” program, we need a printf. And, it is not easy to write a printf.

So, to be able to effectively use a programming language, you need a basic library to be available. And, any language specifies what is a library which every compiler on must provide, and that library is known as a standard library. That library has all the function definition, function names, the hidden names and all of them, fixed by the language design. C has a standard library. As we will see as we move to C++, we will see that C++ has a much stronger standard library. But, C standard library also is quite powerful and interesting. The whole of the C standard library is also available in C++. So, it is not that in C++ things will get replaced; only new things will get added.

Now, C standard library has many components. Every components comes under one header file. And there are, you can look up the total list. But, I have just put in the 5, which are most frequently used. The input output header; standard library header, which has a mix of things to do memory allocation, conversion, searching, sorting and so on.

The string for manipulating C strings, the math library for all different kinds of common mathematical functions and the error header which deals with the error, different error numbers and error ranges and so on. So, please familiarize yourself more with the standard library. And, I would suggest that whenever you are using some standard library component, please look up the manual to see what all other functions that component has. And, it will really, it might so happen that in many places, you are writing certain functions for doing a task, which is already available in the standard library.

(Refer Slide Time: 21:33)

Source Organization for a C program

Header Files

- A header file has extension .h and contains C function declarations and macro definitions to be shared between several source files
- There are two types of header files:
 - Files that the programmer writes
 - Files from standard library
- Header files are included using the `#include` pre-processing directive
 - `#include <file>` for system header files
 - `#include "file"` for header files of your own program

NPTEL MOOCs Programming in C++ Partha Pratim Das 27

At the end, I should also mention that a C program needs to be properly organized. And, it is good to separate between header files and source files, as I said, like the way the library is organized. When you include stdio.h, you are actually including the function prototypes or function headers. The implementations are given somewhere else.

So, when you make use of certain functions of your own, you should also separate them in terms of header, and the files that implement those functions. So, in your program you will have typically have two kinds of header files; the header files that you have written and the header files that the stand has come from the standard library. So, the header files that comes from the standard library or system header files, must come, must be specified with these kinds of angular brackets.

And, the header files written by you must be within double quotes. We will explain the reason for this more, when we go deeper into C++. This is basically, tells the system as to where to look for this file; whether to look for them in system directories or you have to look for them in your current (Refer Time: 22:52) directory.

(Refer Slide Time: 22:54)

Source Organization for a C program

Module 01
Partha Pratim Das

Objectives & Outline

- Recap of C
- Data Types
- Variables
- Literals
- Operators
- Expressions
- Statements
- Control Flow
- Arrays
- Structures
- Unions
- Pointers
- Functions
- Input / Output
- Std Library
- Organization
- Build Process
- References
- Summary

Example:

```
// Solver.h -- Header files
int quadraticEquationSolver(double, double, double, double*, double*);

// Solver.c -- Implementation files
#include "Solver.h"

int quadraticEquationSolver(double a, double b, double c, double* r1, double* r2) {
    // ...
    // ...
    // ...
    return 0;
}

// main.c -- Application files
#include "Solver.h"

int main() {
    double a, b, c;
    double r1, r2;
    int status = quadraticEquationSolver(a, b, c, &r1, &r2);
    return 0;
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 28

So, this is a typical example. I am just trying to deal with an application that needs quadratic equations to be solved. So, there will be several context of why you need to solve the quadratic equation. It could be computing interest; it could be solving some bridge equations, whatever.

So, this is the application program, which assumes that you have a quadratic equation solver. So, you are implementing that. So, you have one header file Solver dot h, which gives a interface of that function or the prototype of the function with all different parameters. You have another, where you put the code of this function or the implementation of the function body. So, you call this Solver dot h; you call this Solver dot c. Solver dot h has all the details. So, here formal parameters have name; here formal parameters do not have name. They are just types given.

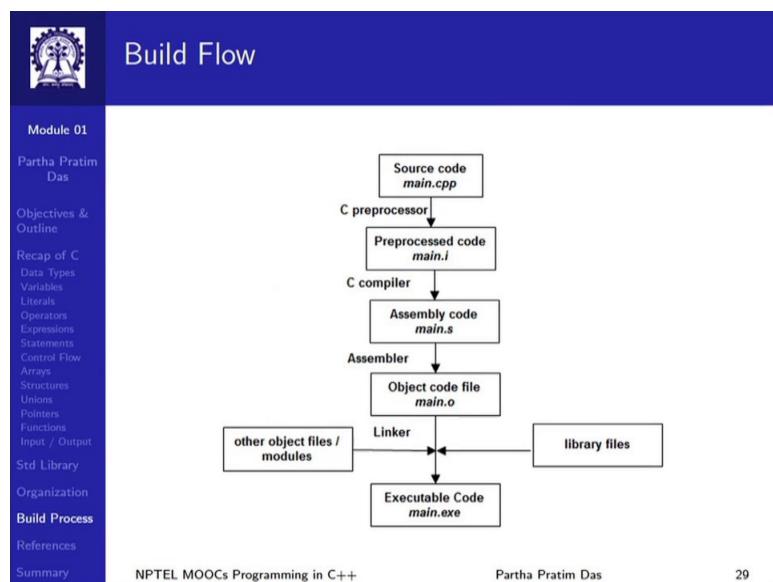
And, in the application you just need to know the header. You do not need to know how

the quadratic equation is solved. All that you need to know is in the signature it has five parameters; first three of them are the three coefficients of the quadratic equation and the next two are the two pointers or two address locations, where the two parts of the result needs to be stored, in case the quadratic equation is solved with a complex value result. So, you need the real and imaginary part to be stored.

So, you just need to know this information to be able to write the application. So, your application file includes Solver dot h, just the header. Your implementation of the quadratic equation solver also includes Solver dot h, which makes sure that, between this solver and your application, the same header is included. So, there is no chance of a mistake. So, you should always separate this out that headers which are common functions, macros, constants, manifest constants that are to be shared between different source files.

Then for every function or for group of functions, you have separate implementation files, which will include the same header. And, this application files will only include the headers. And, we will not need the implementation files to be referred. So, please follow this organization. And, we will see more of that as we go to C++.

(Refer Slide Time: 25:22)



In a typical situation, this is how you prepare your code for execution. You start with a source code, which could be C or C++. Then as you compile, you may compile through an IDE. Just say built or compile something like that or you could do it through a text based interface like you say g plus plus or some C++, something like that. This is the process stages that happen. And, this is just the outline. We will discuss more as we go into the C++ stages.

Your file, your source file first goes through C preprocessor, which takes care of hash include, hash define, this kind of preprocessor directives, then it gets compiled into some kind of a low level language known as assembly language. You see those files have dot s or may be some other systems will have different kinds of file name extensions. Then you assemble them to generate the object files which are called dot o or dot obj. And, there could be several other object files that come from your libraries, like if you have included stdio.h.

Then, that stdio.h gets included at this stage, but the actual implementation is not available. So, that has already been compiled through this process separately and a dot o file has been created. And that dot o file is getting, as you say, linked to your program. If you are using an IDE, you will not be able to see this explicitly because IDE sets it up for you. If you are doing it from the command line, you will have to actually use minus l and include that standard library. And, all of these together will give you the executable code. And, as you go more into C++ we will see that how the built process also impacts the C++ programming and the different programming features.

(Refer Slide Time: 27:31)

The slide has a dark blue header with the title 'Tools'. On the left, there is a vertical sidebar with a logo at the top and a list of navigation links. The main content area contains a bulleted list of tools.

Navigation Links (Sidebar):

- Module 01
- Partha Pratim Das
- Objectives & Outline
- Recap of C
 - Data Types
 - Variables
 - Literals
 - Operators
 - Expressions
 - Statements
 - Control Flow
 - Arrays
 - Structures
 - Unions
 - Pointers
 - Functions
 - Input / Output
- Std Library
- Organization
- Build Process
- References
- Summary

Main Content:

- Development IDE: Code::Blocks 16.01
- Compiler: -std=c++98 and -std=c99

NPTEL MOOCs Programming in C++ Partha Pratim Das 31

In terms of tools, there are multiple IDEs available, like Code::Blocks. There is Visual Studio. There is Eclipse. So, we will advise that you use some IDE, which is open source. Typically, Code Block or Eclipse. And, whatever IDE you are using, you specify that these are the standards; like C 99 is a standard that you are using. Otherwise, the behavior of the programs that we, as we show them and the behavior of the programs as we will experience in the IDE will be different. So always use, for C you use C 99; for C++ we will use C++ 98. Or, we will see, in some context we will use C++ 03. We will come to that.

(Refer Slide Time: 28:26)

This slide is titled 'References' and features a sidebar with a logo and navigation links for Module 01. The main content lists two books:

- Kernighan, Brian W., and Dennis M. Richie. The C Programming Language. Vol. 2. Englewood Cliffs: Prentice-Hall, 1988.
- King, Kim N., and Kim King. C programming: A Modern Approach. Norton, 1996.

A circular profile picture of Partha Pratim Das is on the right.

These are the reference books; in case you want to refer to.

(Refer Slide Time: 28:33)

This slide is titled 'Module Summary' and features a sidebar with a logo and navigation links for Module 01. The main content lists five bullet points summarizing the module's content:

- Revised the concept of variables and literals in C
- Revised the various data types and operators of C
- Re-iterated through the control constructs of C
- Re-iterated through the concepts of functions and pointers of C
- Re-iterated through the program organization of C and the build process.

A circular profile picture of Partha Pratim Das is on the right.

And, so in summary in this whole module we have achieved this. We have revised the concepts of fundamental concepts in C; variables, data types, literals, control constructs. And, we have iterated through functions and pointers and the organization and the build

process.

So, I would expect that you, this will give a quick recap to your fundamentals in C. And, if you have found certain points that you did not understand well or the examples were not well absorbed, then please go through them again or ask questions to us on the blog. But, with this module we will expect that this is a level of C that you are prepared with to be able to proceed with the C++ language training.

We will close module one here. In module two onwards, we will start showing you how in C++ some of the common examples that we have seen here or some of the other common examples in C can be done more efficiently, effectively and in a more robust manner in C++.

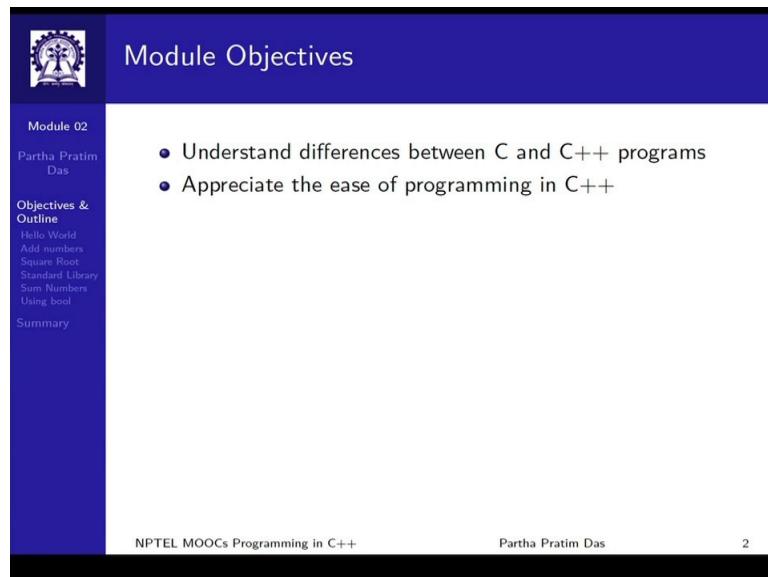
Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 04
Programs with IO and Loop

We will now discuss the module 2, the second module in the programming in C++ course.

In the first module, we have made a revisit of C programming language. We have recapitulated the concepts, the common concepts in C and made sure that we are prepared to slowly get familiar with C++. In this module and in the next 3 modules, we will talk about various programming examples and show, how in C++, this program can be written more efficiently more elegantly and often with better ease, than what is needed in C. So, in module 2, we will get started with understanding the basic differences between C and C++ programs.

(Refer Slide Time: 01:16)



The slide has a dark blue header bar with the title "Module Objectives". On the left, there is a sidebar with the IIT Kharagpur logo, the text "Module 02", "Partha Pratim Das", and a "Objectives & Outline" section listing: Hello World, Add numbers, Square Root, Standard Library, Sum Numbers, Using bool, and Summary. At the bottom of the sidebar, it says "NPTEL MOOCs Programming in C++". The main content area contains two bullet points under the heading "Module Objectives": "Understand differences between C and C++ programs" and "Appreciate the ease of programming in C++". At the very bottom right of the slide, there is a small number "2".

And we will try to appreciate the ease of programming in C++ as we go over this module as well as the following 3 modules.

(Refer Slide Time: 01:32)

Module Outline

Module 02
Partha Pratim Das

Objectives & Outline
Hello World
Add numbers
Square Root
Standard Library
Sum Numbers
Using bool
Summary

- Contrast differences between C and C++ programs for:
 - I/O
 - Variables
 - Using math library
 - Standard Library – Headers
 - Loop
 - bool type

NPTEL MOOCs Programming in C++ Partha Pratim Das 3

We will primarily talk about the contrast in the areas of IO variables, math library standard library headers, loop and the bool type.

(Refer Slide Time: 01:43)

Program 02.01: Hello World

Module 02
Partha Pratim Das

Objectives & Outline
Hello World
Add numbers
Square Root
Standard Library
Sum Numbers
Using bool
Summary

C Program	C++ Program
<pre>// FileName:HelloWorld.c: #include <stdio.h> int main() { printf("Hello World in C"); printf("\n"); return 0; }</pre>	<pre>// FileName:HelloWorld.cpp: #include <iostream> int main() { std::cout << "Hello World in C++"; std::cout << std::endl; return 0; }</pre>
Hello World in C	Hello World in C++
<ul style="list-style-type: none">IO Header is <code>stdio.h</code><code>printf</code> to print to consoleConsole is <code>stdout</code> file<code>printf</code> is a variadic function<code>\n</code> to go to the new line<code>\n</code> is escaped newline character	<ul style="list-style-type: none">IO Header is <code>iostream</code><code>operator<<</code> to <code>stream</code> to consoleConsole is <code>std::cout</code> <code>ostream</code> (in <code>std</code> namespace)<code>operator<<</code> is a binary operator<code>std::endl</code> (in <code>std</code> namespace) to go to the new line<code>std::endl</code> is stream manipulator (newline) functor

NPTEL MOOCs Programming in C++ Partha Pratim Das 4

So, we start with the same initial program in C that is to print “Hello World”. So, here in two columns we show the program to print “Hello World” in C as well as in C++. You

can note some of the basic differences, first is the IO header has changed. In C, it was stdio.h ; in C++ it becomes IO stream. In C, when we do printf, we write to console; we print to console.

Here, we use an operator, a pair of left arrow operators called the output streaming operator to stream to control and the console was stdout file in C, now it is a cout stream in C++. Also note that we are using a prefix before cout that is called std and it is written with std :: . This notation we will get used to quickly, this std is called a namespace; the standard namespace. So, any standard library symbol in C++ will be prefixed with this a particular prefix stream std.

Now, another major point to note in this simple ‘hello world’ program is when we do printf, we can have arbitrary number of parameters, we call this variadic function. So, here in the ‘hello world’ example we are seeing two use of printf, both of which use one parameter, the format stream. Of course, we do not have a format here; it is a constant string being printed. In contrast, in C++ the output streaming operator is a binary operator which as the stream on the left hand side and the content to print on the right hand side and it prints in this form.

So, if we look at the first output line, std :: cout output operator and within quotes, we have a constant string it means that the hello world in C++ string will be streamed to the console cout. Also note that, the new lined character which was escape character back slash n in C. In C++ the same can be used, but there is another way to go to new line that is called endl, which is a short form of end line and we learn later on that. It, endl is basically a stream manipulated. So, this step we are trying to observe that the basic output system in C++ program can be done using cout and the output streaming operator.

(Refer Slide Time: 05:04)

The slide features a header 'Program 02.02: Add two numbers' with a logo on the left. On the left side, there's a sidebar with navigation links: Module 02, Partha Pratim Das, Objectives & Outline, Hello World, Add numbers, Square Root, Standard Library, Sum Numbers, Using bool, and Summary. The main content area is divided into two columns: 'C Program' and 'C++ Program'. Both columns show the code, sample input, and output. Below the code, there's a list of bullet points for each language.

C Program	C++ Program
<pre>// FileName:Add_Num.c #include <stdio.h> int main() { int a, b; int sum; printf("Input two numbers:\n"); scanf("%d%d", &a, &b); sum = a + b; printf("Sum of %d and %d", a, b); printf(" is: %d\n", sum); return 0; }</pre>	<pre>// FileName:Add_Num_c++.cpp #include <iostream> int main() { int a, b; std::cout << "Input two numbers:\n"; std::cin >> a >> b; int sum = a + b; // Declaration of sum std::cout << "Sum of " << a << " and " << b << " is: " << sum << std::endl; return 0; }</pre>
Input two numbers: 3 4 Sum of 3 and 4 is: 7	Input two numbers: 3 4 Sum of 3 and 4 is: 7
<ul style="list-style-type: none">• scanf to scan (read) from console• Console is stdin file• scanf is a variadic function• Addresses of a and b needed in scanf• All variables a, b & sum declared first (C89)• Formatting (%d) needed for variables	<ul style="list-style-type: none">• operator>> to stream from console• Console is std::cin istream (in std namespace)• operator>> is a binary operator• a and b can be directly used in operator>>• sum may be declared when needed• Formatting is derived from type (int) of variables

NPTEL MOOCs Programming in C++ Partha Pratim Das 5

We move to the next program, where we are illustrating a very simple arithmetic program, which has two variables, a and b and adds them to generate their sum. The values of these variables are read from the console again, which in C is std in and we use the scanf function, which all of you are familiar with the format string scanf, like printf is a variadic function that is, it takes variable number of parameters. Here, we see a form of scanf which is taking 3 parameters; the format string and the addresses of a and b, respectively.

In contrast, in C++ program we introduce another operator which is used for streaming from an input stream. So, this is called an input streaming operator, It is again a pair of arrows, but the arrows now direct from left to right. So, if you look into std ::cin, the input operator then a it means that is being read by C in. What is interesting is, in this format in C++ you can actually put multiple operators multiple operands multiple variables one after the other as we are showing here.

So, after streaming a, from the standard input we again, stream b from the standard input. So, this notation means that first a and then variable b will be read from the standard input of the console. We also show how these variables are output to the standard streaming output which is cout or std out as in C program.

The two major differences are, that you must note here is, one we do not need to use the format string in C++. In C we know, if I want to print an integer variable I need to specify in the format string that the variable is to be printed in percentage d format, which denotes that is an integer type of data to be printed, In C++, in contrast the variable does not need to be specified with a format, the compiler knowing that it is an integer variable will automatically decide the format that is required for printing it and print it in the right way.

A second major difference that you should note in between these two programs is when we do scanf, we are reading values from the ‘std in’ and since in reading that value the value original value of variable a has to be changed to the value that is input by the user. We need to pass the address of a as it would be familiar in C this is kind of a call by address mechanism that is being used, where we pass the address of the variable as a call by value parameter to the scanf function.

In contrast, in C++ when we are reading from the input stream, we do not need the address to be passed; we can simply specify the variable and the compiler will take care of the rest. When we learn more of the C++ operators and call mechanisms, we will understand how this really works, but it certainly becomes more convenient to get rid of the format string as well as the need to specify either the variable for printf or the address of the variable for scanf in C++ everything can be done uniformly.

There is another small difference that you may note in terms of the declaration of variable sum, in the C program the variable sum is declared at the top after the variables a and b, because the original C or the old version of C, which is C89 specified that all declaration of variables must happen before the first executable statement in the program.

In the C program that we see here in the first executable is the printf function call. So, all variables must be declared before that. In C++ this restriction does not exist. So, we can declare the variable as and when we need them when we are starting the name we just need the variables a and b because there need to be read, but when we need to do their sum at that point we can declare sum as a variable, and then use a plus b and put that

result into sum to initialize the value of sum. Of course, this should be noted that later version of C that is C99 does allow you to defer the declaration of variables to like in C++ to the point when you actually need the variable. So, please carefully read and run this program at your computer to understand the similarity and the differences better.

(Refer Slide Time: 11:01)

The screenshot shows a slide titled "Program 02.03: Square Root of a number". The slide has a sidebar on the left with the NPTEL logo and navigation links for Module 02, Partha Pratim Das, Objectives & Outline, Hello World, Add numbers, Square Root, Standard Library, Sum Numbers, Using bool, and Summary. The main content area compares two programs:

C Program	C++ Program
<pre>// FileName:Sqrt.c #include <stdio.h> #include <math.h> int main() { double x; double sqrt_x; printf("Input number:\n"); scanf("%lf", &x); sqrt_x = sqrt(x); printf("Sq. Root of %lf is:", x); printf("%lf\n", sqrt_x); return 0; }</pre> <p>Input number: 2 Square Root of 2.000000 is: 1.414214</p>	<pre>// FileName:Sqrt_c++.cpp #include <iostream> #include <cmath> using namespace std; int main() { double x; cout << "Input number:" << endl; cin >> x; double sqrt_x = // Declaration of sqrt_x sqrt(x); cout << "Sq. Root of " << x; cout << " is: " << sqrt_x << endl; } return 0; }</pre> <p>Input number: 2 Square Root of 2 is: 1.41421</p>
<ul style="list-style-type: none"> • Math Header is <code>math.h</code> (C Standard Library) • Formatting (<code>%lf</code>) needed for variables • <code>sqrt</code> function from C Standard Library • Default precision in print is 6 	
<ul style="list-style-type: none"> • Math Header is <code>cmath</code> (C Standard Library in C++) • Formatting is derived from type (<code>double</code>) of variables • <code>sqrt</code> function from C Standard Library • Default precision in print is 5 (different) 	

At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" with a page number "6".

We will next move on to another program, which is again a simple program using mathematical computation, which I am sure you have done at some points of time in the C program. So, we know that in C there is a `math.h` header as a part of the C standard library, which has a number of useful functions. So, here we are just illustrating the use of one such function called `sqrt`, for finding the square root of a double variable. It takes a double variable and returns a double result which is a square root of the parameter that is passed to `sqrt`.

The similar function can be invoked in C++ as well. So, we are showing how to do that, you please note that the header that we use in C++ now as changed in its name. In C we are calling it `math.h` the same header, in C++ is called `cmath` and we will see this is a common convention that any C standard library header can be used in C++ as well, but when you use that, you add a C at the beginning of the name. The C means that the

standard library header is coming from the C standard library and the other difference is you drop the dot h extension to the file name that existed in C, you just call it cmath.

Of course, when we do this, as I had mentioned earlier in terms of the names of cout and endl, these also are in the same namespace of std, which means that the function sqrt in C is just called sqrt, whereas in C++ this function name will become sqrt prefixed with std, that is full name is std ::sqrt. Now, here we also show another short cut or convenient way to express the standard library symbols, note after the hash includes in the C++ program, we have written a line saying using namespace std. This means that if we include this line then any standard library symbol that occurs in the C++ program will be assumed to have std:: as a prefix, we will not have to every time write std :: cout, std :: cin, std :: endl or std :: sqrt.

So, this is a convenient way of doing that you could either make use of using namespaces feature in C++ or if you are not using it then all standard library symbols will need to be prefixed with std:: . Rest of the program is very easily understandable and it is pretty much like the C program that you see on the left or the changes are according to the input and the output streaming as we have already seen. So, with this we will move on and take a look into the C++ standard library.

(Refer Slide Time: 14:31)

The slide has a dark blue header with the title 'namespace std for C++ Standard Library'. On the left, there's a sidebar with a logo, the text 'Module 02', 'Partha Pratim Das', 'Objectives & Outline' (listing 'Hello World', 'Add numbers', 'Square Root', 'Standard Library', 'Sum Numbers', 'Using bool', 'Summary'), and a circular profile picture of Partha Pratim Das.

The main content area is divided into two columns: 'C Standard Library' and 'C++ Standard Library'.

C Standard Library	C++ Standard Library
<ul style="list-style-type: none">All names are global• std::cout, std::stdin, printf, scanf	<ul style="list-style-type: none">All names are within std namespace• std::cout, std::cin• Use <code>using namespace std;</code> to get rid of writing std:: for every standard library name

Below this, there are two sections: 'W/o using' and 'W/ using'.

W/o using	W/ using
<pre>#include <iostream> int main() { std::cout << "Hello World in C++" << std::endl; return 0; }</pre>	<pre>#include <iostream> using namespace std; int main() { cout << "Hello World in C++" << endl; return 0; }</pre>

At the bottom, the footer says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

Just summarizing, what I have already specified that in a C standard library all names are global, that is all standard library functions, macros; they are available to any function by the name that they have. So, therefore, all C standard library names are actually for all practical purposes get reserved in a way that is you cannot write a printf function of your own and also continue to use the printf function given in the C standard library because the moment you write a printf function of your own, that also as a name in the global space.

You will understand this more when we study about namespaces formally in C++, but please note that all names are just available in the global space. In contrast, in C++ standard library all names are in the std namespace. This is specifically reserved for the standard library this namespaces and all names are prefixed with std:: meaning that the name occurs within this std namespaces. So, namespace is like when we use in our own names, it is like a family name or the last name that we use. So, I am Partha Pratim Das. So, Das is my last name and my name is Partha. So, there could be another Partha in some other family names, say Partha Pratim Chakrabarti. So, these are distinguished by the different family names that exist.

So, names space is something similar to that. We will talk more about it subsequently. So, we just show also illustrate here that if you do the short cut of putting using namespaces std, then you do not need to prefix all standard library names with that std :: namespace.

(Refer Slide Time: 16:38)

The slide has a dark blue header with the title "Standard Library Header Conventions". On the left, there is a sidebar with a logo, the name "Partha Pratim Das", and a "Module 02" section. Below that are links to "Objectives & Outline", "Hello World", "Add numbers", "Square Root", "Standard Library", "Sum Numbers", "Using bool", and "Summary". There is also a circular profile picture of Partha Pratim Das.

The main content area contains a table comparing C and C++ header conventions:

	C Header	C++ Header
C Program	Use .h Example: #include <stdio.h> Names in global namespace	Not applicable
C++ Program	Prefix c. no .h Example: #include <cstdio> Names in std namespace	No .h Example: #include <iostream>

Below the table, there are two bullet points:

- Any C standard library header is to be used in C++ with a prefix 'c' and without the .h. These symbols will be in std namespace. Like:
`#include <cmath> // In C it is <math.h>
...
std::sqrt(5.0); // Use with std::`
- It is possible that a C++ program include a C header as in C. Like:
`#include <math.h> // Not in std namespace
...
sqrt(5.0); // Use without std::`
This, however, is not preferred.

At the bottom, the footer says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" with a page number "8".

Now, I would like to highlight something, which is very specific about headers in of standard library. So, we have noted that C++ is backward compatible to C what does that means; it means that any C program should be executable as a C++ program also. This brings in another question that what do we do with the standard library of C, as I have already illustrated that standard library of C can also be used in a C++ program, but there is a small point to be noted by you mix the program from C with the program from C++ in terms of how you specify the standard library headers.

So, this table show you how you can do this on the left, on the rows we show language in which you are writing the program and on the column we are showing the header being used from which standard library, whether it is from C or it is from C++. So, if you are writing a C program and you are using C standard library headers, we all know you will include like stdio.h. If you are writing a C++ program and including a C standard library header then as I have mentioned, you will need to prefix the C standard library name with a C. So, stdio.h now becomes C stdio and you will have to drop the dot h from the name and all of these symbols from the C standard library now gets into the std namespace and we will have to be prefixed with std :: .

If you write a C++ program and include the C++ standard library, you will simply include it as hash include IO stream. All standard library headers in C++ do not have any dot h in their file extension, this is for a historical reason which I would try to explain at a later point of time, but please note that IO stream.h should not be included and the last box in this matrix that if you are writing a C program and you want to use a C++ header certainly is not applicable because you cannot do that C++ has a lot of features which C does not support and therefore, such a use cannot be done.

Specifically note and I have highlighted with red that by mistake or by the practice of using dot h as a file name extension for standard library headers in C. If you include IO stream.h in a C++ program, your compiler may not actually give you an error which means that your compiler is actually dated and you should move to a more recent compiler and therefore, this is a very dangerous preposition because you are making a mistake IO stream.h or for that matter any C++ standard library header with dot h extension files, all of them have been depicted. They are no more in use, but some compiler still continue to allow them because they were written before all these changes were done in the C++ standard. So, please keep in mind these conventions of standard library headers.

(Refer Slide Time: 20:26)

The slide title is "Program 02.04: Sum n natural numbers". The left sidebar shows "Module 02" and "Partha Pratim Das". The main content area compares two programs:

C Program	C++ Program
<pre>// FileName:Sum_n.c: #include <stdio.h> int main() { int n; int i; int sum = 0; printf("Input limit:\n"); scanf("%d", &n); for (i = 0; i <= n; ++i) sum = sum + i; printf("Sum of %d", n); printf(" numbers is: %d\n", sum); return 0; }</pre>	<pre>// FileName:Sum_n.cpp: #include <iostream> using namespace std; int main() { int n; int sum = 0; cout << "Input limit:" << endl; cin >> n; for (int i = 0; i <= n; ++i) // Local Decl. sum = sum + i; cout << "Sum of " << n ; cout << " numbers is: " << sum << endl; return 0; }</pre>
Input limit: 10 Sum of 10 numbers is: 55	Input limit: 10 Sum of 10 numbers is: 55
<ul style="list-style-type: none"> • i must be declared at the beginning (C89) • i declared locally in for loop 	

At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

Next, we will look into the use of loops which is very similar to what you we have in C. So, here we are just adding a set of a sequence of numbers starting from 0 up to n and summing them using a for loop. The same program almost identically will work for C++ except for the differences in the IO headers and the cout streaming convention. You may also note that for loop the loop index, i can be declared within the parenthesis ‘for’ construct.

If you do that then this declaration of i is local to this for loop that is once you come out of the for loop you are in the subsequent cout statement or later on i will not be considered to have been declared. So, this was introduced, so that you can just whenever you need local index variables you could quickly locally use them and do not have to really think about whether you have declare that variable earlier whether it is being used in some other context and so on. You can just locally declare them and use them in C++. This was not possible in C89, but this is now possible in C99 also.

(Refer Slide Time: 22:02)



Program 02.05: Using bool

Module 02	C Program	C++ Program
Partha Pratim Das Objectives & Outline Hello World Add numbers Square Root Standard Library Sum Numbers Using bool Summary	<pre>// FileName:bool.c: #include <stdio.h> #define TRUE 1 #define FALSE 0 int main() { int x = TRUE; printf ("bool is %d\n", x); return 0; } bool is 1</pre> <ul style="list-style-type: none"> • Using int and #define for bool • May use _Bool (C99) 	<pre>// FileName:bool.c: #include <stdio.h> #include <stdbool.h> int main() { bool x = true; printf ("bool is %d\n", x); return 0; } bool is 1</pre> <ul style="list-style-type: none"> • stdbool.h included for bool • _Bool type & macros (C99): bool which expands to _Bool true which expands to 1 false which expands to 0
		<pre>// FileName:bool_c++.cpp: #include <iostream> using namespace std; int main() { bool x = true; cout << "bool is " << x; return 0; }</pre> <ul style="list-style-type: none"> • No additional headers required bool is a built-in type true is a literal false is a literal



NPTEL MOOCs Programming in C++ Partha Pratim Das 10

Finally, in the last section of this module, we illustrate the use of the Boolean type. We all know that C has a possible use of a Boolean type, which is a type where we say that it can take a value, either true or false. Now, C which is C89, the original old C that we have did not have a separate type for bool. So, what it did is it was using the int to

interpret bool that is anywhere you want to put a Boolean condition or a Boolean value you will declare an int variable and set that to 0, if you want to mean false and set that to something non-zero, if we want to mean true.

So, out of these 3 columns if you look at the left most columns is the most common way that C programs had been dealing with the Boolean, you could for convenience define two constant; true and false to be 1 and 0 and use them in your program, but as I show the int variable to be the variable x to be used for the Boolean condition is declared as of int type and is initialized with true. So, if you print it will show that it has a value 1; this is what existed in C89. Subsequently in C99, a change has been made to introduce a bool type.

Now, before looking into that let us first look into the right most columns, which is the C++ program. In C++ you have bool as a built-in type as you have int, char, float, double. Similarly, you have a bool type, this bool type as only two literals; true and false both in lower case. So, they are keywords now reserved as well. So, you want to similarly define x for use as a Boolean variable, you can directly use bool which mix it very easy to understand that you are actually dealing with a Boolean value and you could initialize it with true or false and, but if you try to print the value of this variable, it will not print true or false, it will actually print 1 or 0. it is 1, if it true and 0, otherwise.

Naturally, being able to use and explicit built-in type bool has a lot of advantages. The most significant of that is from the C++ program, if you have used bool to specify your Boolean value, anyone else who reads the program will be able to understand that this variable cannot take any other value, other than true or false. In contrast, if we use the C style of using int to mean Boolean value, then it can actually take multiple different values which are interpreted as true or false, as a case may be.

Now, in the middle column, we see an interesting extension of C programming language in C99 standard what C99 came up with. C99 introduced an explicit Boolean type and that is given by the name underscore bool, where b is written in capital, but that since that is not a very normal natural way to write bool, it has also provided a new standard library header called stdbool.h, where three macros are provided.

The first macro defines bool in lower case as same as underscore capital bool. So, if we use the bool in a lower case in a C99 program, then you are actually using that new predefined type underscore capital bool and it also defines true and false as 1 and 0 in the header, so that you could use them as constant here. So, if we are; whenever you are using C, now you should always the bool type and not use int and interpret it as a bool type. In C++ certainly it is come out to be a built-in type. So, we show that it has several other advantages as well as we go along with the different types.

(Refer Slide Time: 26:54)

The slide is titled "Module Summary" and features a blue header bar. On the left, there is a sidebar with a logo, the title "Module 02", the name "Partha Pratim Das", and a list of objectives and outline items: Hello World, Add numbers, Square Root, Standard Library, Sum Numbers, Using bool, and Summary. Below the sidebar is a circular portrait of Partha Pratim Das. The main content area contains a bulleted list of three points:

- Understanding differences between C and C++ for:
 - IO
 - Variable declaration
 - Standard Library
- C++ gives us more flexibility in terms of basic declaration and input / output
- Many C constructs and functions are simplified in C++ which helps to increase the ease of programming

At the bottom of the slide, the footer includes the text "NPTEL MOOCs Programming in C++", the name "Partha Pratim Das", and the number "11".

So in this module, we have tried to understand the basic differences between C and C++ with specific focus to; how you perform input output? How you declare variables? And how the standard library of C and C++ are used in C++? We have started to see that C++ gives us more flexibility in terms of how we can declare and how we can do input output.

Now, we do not need to have those complicated printf statements, stream printf function calls where the formats are into a separate strings, variables are listed separately. We do not need to remember that scanf needs the address of variables and so on and in this way many construct and functions have been simplified in C++, which will help in increases ease of programming.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 05
Arrays and Strings

Welcome to module 3 of Programming in C++. This module we will discuss about arrays and strings. We have in module 2 seen what are the basic differences between a C program and a C++ program. We will continue on the same note.

(Refer Slide Time: 00:41)

The slide has a dark blue header with the title "Module Objectives". On the left, there is a sidebar with a logo at the top, followed by "Module 03", "Partha Pratim Das", "Objectives & Outline", and a list of topics: "Arrays & Vectors", "Fixed Size Array", "Arbitrary Size Array", "Vectors", "Strings", and "Summary". Below the sidebar is a circular portrait of Prof. Partha Pratim Das. The main content area contains a bulleted list of objectives:

- Understand array usage in C and C++
- Understand vector usage in C++
- Understand string functions in C and string type in C++

At the bottom of the slide, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" with a page number "2".

In this module we will try to particularly understand the use of arrays in C and C++. We will introduce a basic notion of what is called vector in C++, which is pretty much like arrays, but lot more powerful and we will try to see how strings are used in C and in contrast how the string type operates in C++.

(Refer Slide Time: 01:06)

This slide shows the module outline for NPTEL MOOCs Programming in C++. The title is "Module Outline". The left sidebar contains a navigation menu with "Module 03", "Partha Pratim Das", "Objectives & Outline", "Arrays & Vectors", "Fixed Size Array", "Arbitrary Size Arrays", "Vectors", "Strings", and "Summary". Below the menu is a circular profile picture of Partha Pratim Das. The main content area lists two sections: "Arrays and Vectors" and "Strings in C and C++".

- Arrays and Vectors
 - Fixed size arrays – in C and C++
 - Arbitrary size arrays – in C and C++
 - vectors in C++
- Strings in C and C++
 - string functions in C and C++
 - string type in C++
 - String manipulation in C++

NPTEL MOOCs Programming in C++ Partha Pratim Das 3

So, these are the points that we will cover.

(Refer Slide Time: 01:11)

This slide displays two programs side-by-side: a C program and a C++ program, both titled "Program 03.01: Fixed Size Array". The left sidebar is identical to the previous slide, showing "Module 03", "Partha Pratim Das", "Objectives & Outline", "Arrays & Vectors", "Fixed Size Array", "Arbitrary Size Arrays", "Vectors", "Strings", and "Summary". Below the menu is a circular profile picture of Partha Pratim Das.

C Program	C++ Program
<pre>// File Name:Array_Fixed_Size.c: #include <stdio.h> int main() { short age[4]; age[0] = 23; age[1] = 34; age[2] = 65; age[3] = 74; printf("%d ", age[0]); printf("%d ", age[1]); printf("%d ", age[2]); printf("%d ", age[3]); return 0; }</pre>	<pre>//FileName:Array_Fixed_Size_c++.cpp: #include <iostream> int main() { short age[4]; age[0] = 23; age[1] = 34; age[2] = 65; age[3] = 74; std::cout << age[0] << " "; std::cout << age[1] << " "; std::cout << age[2] << " "; std::cout << age[3] << " "; return 0; }</pre>
23 34 65 74	

• No difference between arrays in C and C++

NPTEL MOOCs Programming in C++ Partha Pratim Das 4

To get started we have side by side shown two programs, both of them. They actually are identical except for the difference in the IO header. So, a basic use of array as in C can be

done in C++ exactly in the same notation and with the same meaning. So, the first message is arrays can be used in C++ exactly as you know in C. So, here we are just assigning some values to the different array elements and printing them, only difference is in terms the using the printf or using the cout.

(Refer Slide Time: 01:56)

Arbitrary Size Array

This can be implemented in C (C++) in the following ways:

- **Case 1:** Declaring a large array with size greater than the size given by users in all (most) of the cases
 - Hard-code the maximum size in code
 - Declare a manifest constant for the maximum size
- **Case 2:** Using `malloc (new[])` to dynamically allocate space at run-time for the array

Module 03
Partha Pratim Das
Objectives & Outline
Arrays & Vectors
Fixed Size Array
Arbitrary Size Array
Vectors
Strings
Summary

NPTEL MOOCs Programming in C++ Partha Pratim Das 5

Now, the one of the main issues in C, in terms of using arrays that you all must have faced is, when I want to use an array in C, I need to know; I need to specify the size of the array which means at the maximum number of elements that the array can contain beforehand, that is at the time of writing the program or to be specific at the time of compiling the program. So, if I do not know that size, then I need to provide a size which is greater than what can happen at in any one of the cases that I execute, run in the program.

Certainly there are two ways of handling this situation. One is, I define, declare a large array and this can be done; this can be either hard coded, the size can be hard coded within the program or the size can be somewhat soft coded by using a manifest constant and the other case that you have seen earlier in C programming is you would use malloc to dynamically allocate space and allocate the array at the run time and then you use it and if you are using malloc, you will also have to remember to free it up, when you are use of that array is done.

(Refer Slide Time: 03:21)

The slide shows two versions of a C program for calculating the sum of elements in an array. The left column, titled 'Hard-coded', contains the original code where the array size is hard-coded as 100. The right column, titled 'Using manifest constant', shows the same code but uses a manifest constant MAX defined as 100. Both programs prompt the user for the number of elements and the sum of the array. The output for both cases shows the input 'Enter no. of elements: 10' and the output 'Array Sum: 45'. A note at the bottom of each column indicates the advantage of using a manifest constant.

	Hard-coded	Using manifest constant
// FileName:Array_Large_Size.c:	// FileName:Array_Macro.c:	
#include <stdio.h>	#include <stdio.h>	
#include <stdlib.h>	#include <stdlib.h>	
	#define MAX 100	
int main() {	int main() {	
int arr[100], sum = 0, i;	int arr[MAX], sum = 0, i;	
printf("Enter no. of elements: ");	printf("Enter no. of elements: ");	
scanf("%d", &count);	scanf("%d", &count);	
for(i = 0; i < count; i++) {	for(i = 0; i < count; i++) {	
arr[i] = i;	arr[i] = i;	
sum += arr[i];	sum += arr[i];	
}	}	
printf("Array Sum: %d", sum);	printf("Array Sum: %d", sum);	
return 0;	return 0;	
Enter no. of elements: 10	Enter no. of elements: 10	
Array Sum: 45	Array Sum: 45	
• Hard-coded size	• Size by manifest constant	

NPTEL MOOCs Programming in C++ Partha Pratim Das 6

So, let us see, how does this look in terms of as we migrate from C to C++. So, this is just showing the C example on the left hand side. We are hard coding the size of the array arr to 100 on the right column we are doing the same thing, except that now you have a manifest constant MAX, which is defined to have a value 100 and we are using that MAX. The advantage of using the manifest constant is there could be 10 different arrays, whose size is to be specified to 100.

Now, if I hard code all of them and sometime later I need to change all those sizes from 100 to 1000, then at 10 places I will need to go and edit and I might just forget to do all of them, but if I hash define or use a manifest constant then I can make the change only at one place, change the definition of MAX 100 to 1000 and all of those will change. So, it is a better programming practice in C to use manifest constant and not hard code value, you already know this.

(Refer Slide Time: 04:35)



Program 03.03: Fixed large array / vector

Module 03	C (array & constant)	C++ (vector & constant)
Partha Pratim Das Objectives & Outline Arrays & Vectors Fixed Size Array Arbitrary Size Array Vectors Strings Summary	<pre>// FileName:Array_Macro.c: #include <stdio.h> #include <stdlib.h> #define MAX 100 int main() { int arr[MAX], sum = 0, i; printf("Enter no. of elements: "); int count; scanf("%d", &count); for(i = 0; i < count; i++) { arr[i] = i; sum += arr[i]; } printf("Array Sum: %d", sum); return 0; }</pre> <p>Enter no. of elements: 10 Array Sum: 45</p>	<pre>// FileName:Array_Macro_c++.cpp: #include <iostream> #include <vector> using namespace std; #define MAX 100 int main() { vector<int> arr(MAX); // Define-time size cout << "Enter the no. of elements: "; int count, j, sum = 0; cin >> count; for(int i = 0; i < count; i++) { arr[i] = i; sum += arr[i]; } cout << "Array Sum: " << sum << endl; return 0; }</pre> <p>Enter no. of elements: 10 Array Sum: 45</p>
	<ul style="list-style-type: none"> • MAX is the declared size of array • No header needed • arr declared as int [] 	<ul style="list-style-type: none"> • MAX is the declared size of vector • Header vector included • arr declared as vector<int>
		Partha Pratim Das
		7

Now, you show the similar situation in between C and C++ on the right column now, you have a C++ program. Certainly, the array size, the maximum array size as MAX can be hash define to 100 as has been done, but what we show here is just focus on the line right after the header of main. We are writing vector within corner bracket int and then the array name and within parentheses, we show the size MAX.

Vector is a new introduction in C++. This is not a built-in type; please do not consider this to be a built-in type. This is something which is provided by the C++ standard library. So, if you move your attention to the top in terms of the #include list, you will find #, there is #includ<vector>. So, there is standard library header vector which has all the necessary declaration definitions for a vector type and you can use that in this way; what it means is vector for in all respect what likes arrays can be. So, you just focus within the ‘for’ loop, you see how the array elements are being accessed.

On the left hand side, it is a well known array int arr[MAX]. We write it as arr[i] on the right, it is a vector of int we use the same indexing notation to access the array elements. So, vector is same in terms of access notation and the result of doing the read or write access with the traditional array, but it has the advantage that its size is not necessarily fixed at the compile time.

Now, in this example we have just shown that how to use vector with a fixed initial size MAX. So, when we say vector and within corner bracket int, what we mean that within corner bracket, we are providing the type of the element that the array is composed of, which is what we write in C as int arr[MAX], we write it as vector within corner bracket int and whatever we provide as the maximum size within the square brackets here, we pass it as a parameter after the arr name.

So, this is just a notational difference, right now just accept this as a different notation of writing, declaring arrays and once you have done that rest of the program you can forget about, that you are specifically using a vector you can just continue to use them as arrays.

(Refer Slide Time: 07:42)

C Program	C++ Program
<pre>// FileName:Array_Malloc.c #include <stdio.h> #include <stdlib.h> int main() { printf("Enter no. of elements "); int count, sum = 0, i; scanf("%d", &count); int *arr = (int*) malloc (sizeof(int)*count); for(i = 0; i < count; i++) { arr[i] = i; sum += arr[i]; } printf("Array Sum:%d ", sum); return 0; }</pre>	<pre>// FileName:Array_Resize_c++.cpp #include <iostream> #include <vector> using namespace std; int main() { cout << "Enter the no. of elements: "; int count, j, sum=0; cin >> count; vector<int> arr; // Default size arr.resize(count); // Set resize for(int i = 0; i < arr.size(); i++) { arr[i] = i; sum += arr[i]; } cout << "Array Sum: " << sum << endl; return 0; }</pre>
Enter no. of elements: 10 Array Sum: 45	Enter no. of elements: 10 Array Sum: 45
<ul style="list-style-type: none"> • malloc allocates space using sizeof • resize fixes vector size at run-time 	

NPTEL MOOCs Programming in C++ Partha Pratim Das 8

Now with this, let me show where you actually get the advantage. Now, let us focus on the second mechanism of using arbitrary sized arrays that is you do not know at all as to how large an array can be you will get to know only when the program is executed by the user. So, the user will probably provide the size of the array, the number of elements that the user wants.

So, on the C program, on the left see that the first we are asking the user; how many elements

are there, and the user provides a count value and if you have to do this in C, then will have to dynamically allocate the array using malloc as is shown and proceed with that. Certainly you will need to write a very complex form in malloc because you really need to say how much memory you want, malloc returns you a void star pointer, you need to remember and cast that to the int star.

All those nuances of C programming exist; now just shift your focus to the right on the same lines. Now, we are declaring the variable as a vector of int the variable is arr and please note in contrast to what we had shown earlier, we are not passing any MAX size. So, if we do not pass a MAX size then we get a vector of a default sizes, C++ standard library will have some default size which is not fixed, but some default size array will be there, but the following line we write something which you are not familiar in the notation. We are writing arr.resize(), we call this as resize being a member function of the vector type. What it does, is in this resize function if we pass a value as we are passing through the variable count, then the vector will resize to the count number of elements.

So, let us assume that the default size with which the vector was created is 10 and now the user at the run time as given an input 100 to count, when arr.resize() is done the value will be passed as 100 and the vector will change to having hundred elements from the original ten elements.

So, resize can be used very conveniently to increase or decrease the number of elements that a vector can have or for that matter the vector form of array can have. So, with that you get rid of using all this malloc and its complicated notation and remembering to free that location and so on. We can just use vector and resize them as needed which makes the use of arrays in C++ programs as vector container is far more convenient and compact than the similar mechanism in C.

(Refer Slide Time: 10:59)

The slide has a dark blue header with the title "Strings in C and C++". Below the header is a sidebar with a logo and navigation links for "Module 03", "Partha Pratim Das", "Objectives & Outline", "Arrays & Vectors", "Fixed Size Array", "Arbitrary Size Vectors", "Strings", and "Summary". The main content area contains a section titled "String manipulations in C and C++:" with two bullet points:

- C-String and `string.h` library
 - C-String is an array of `char` terminated by `NULL`
 - C-String is supported by functions in `string.h` in C standard library
- `string` type in C++ standard library
 - `string` is a type
 - With operators (like `+` for concatenation) behaves like a built-in type

At the bottom left is a circular profile picture of the speaker, Partha Pratim Das. The footer includes the text "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and the number "9".

Next, let us take a look in the handling of strings as you are; would be already familiar that besides 2 numerical types that is whole numbers `int` and the floating point numbers, the next most widely used and most required type or values that we need to deal with our strings where we are talking about a sequence of characters, and what do we have in; if we are working in C, we have what is now called a C string.

C does not have a default type as `string`, but it has a `string.h` standard library header which provides a whole lot of string functions like `strlen`, `strcpy`, `strcat` and so on and with that C string is just an array of characters, which we say is terminated by `NULL`, which means that if you scan the array from left to right, you will continue to consider that you have a string till you come across the first null characters or the characters with ASCII value 0; please note that in the array after this `NULL`, there could be several other characters still remaining, but they are not considered to be part of the string.

Now, with this convention if we use the string functions from `string.h` header then you will be able to achieve variety of string operations as you all are familiar with. In contrast, C++ now introduces a `string` type in C++ standard library. This is pretty much like we talked about vector. So, `string` also is not a built-in type, but it is a type added through the standard library and you will have to use the `string` header of C++ standard library to get the strings and it has

some amazing behavior like being able to write concatenations of string as an addition expression.

(Refer Slide Time: 13:06)

C Program	C++ Program
<pre>// FileName:Add_strings.c: #include <stdio.h> #include <string.h> int main() { char str1[] = {'H','E','L','L','O',' ', '\0'}; char str2[] = "WORLD"; char str[20]; strcpy(str, str1); strcat(str, str2); printf("%s\n", str); return 0; }</pre>	<pre>// FileName:Add_strings_c++.cpp: #include <iostream> #include <string> using namespace std; int main(void) { string str1 = "HELLO "; string str2 = "WORLD"; string str = str1 + str2; cout << str; return 0; }</pre>
HELLO WORLD	HELLO WORLD
<ul style="list-style-type: none"> Need header <code>string.h</code> C-String is an array of characters String concatenation done with <code>strcat</code> function Need a copy into <code>str</code> <code>str</code> must be large to fit the result 	
<ul style="list-style-type: none"> Need header <code>string</code> <code>string</code> is a data-type in C++ standard library Strings are concatenated like addition of <code>int</code> 	
<small>NPTEL MOOCs Programming in C++</small>	
<small>Partha Pratim Das</small>	
<small>10</small>	

So, we will illustrate those. Here is a simple parallel between a C program and a C++ program. This program starts with two strings that are defined within the program the hello world and we want to concatenate the second string after the first string. So, we just want to put them side by side the first string followed by the second string and make one concatenated string.

So, if you have to do that in C, on the left you can see what you will need to do you will need to have an array large enough to contain the concatenated string let us called it str, you will have to copy the first string str1 into str and then you will have to concatenate str2 into what is already copied in str. So, it will just come after that, so first hello will hello followed by a blank will get copied to str and then world will get concatenated strcpy and strcat does the job and then you can print it.

In contrast, in C++ you have a string type in the string header. So, you include the string header now you do not declare them as characters arrays you declare them as string which is

the name of the type given in that header; please note that this name is all in lower case and then you have the string variable, name str1 and you initialize it constant string hello blank or world.

The very interesting things is when you have to concatenate it you do not really need to copy the first string and then do concatenation, you can just say that I am adding str2 to str1. So, we say str1+ str2, so this is pretty much like I have a variable x having value 3, have a variable y having value 5. I write x plus y to mean 3 plus 5, which is 8. So that is an integer addition.

This is kind of a string addition in the type of string, this becomes a concatenation operation and we will see the amazing power in C++ to be able to define operators for your own types in whatever way you want to interpret them. For example, you could use this to write algebra for rectangles you can have two rectangles, if you have a rectangle type and you can define then the addition of two rectangles is basically making a union of this rectangles to make a rectangle large enough to contain both these rectangles and so on. So, this features in terms of string is available in C++ therefore, it becomes really easy to deal with strings.

In C++ particularly note that in C, you will really need to know what is the size of the result? So, that you can define again an array large enough for the variable str because if this is str is not enough in size then strcpy, strcat later on will fail in C++ you do not need to bother about any of this when you do when you declare the variable str as a type string and you initialize it with the concatenation of str1+ str2 the compiler automatically takes care of managing the size and will give you a string which is large enough to contain the concatenation. So, there is a lot of ease in the whole handling of strings.

(Refer Slide Time: 17:00)

The screenshot shows a presentation slide with a dark blue header containing the title 'More on Strings'. On the left, there is a sidebar with a logo at the top, followed by 'Module 03' and 'Partha Pratim Das'. Below this, under 'Objectives & Outline', are several sections: 'Arrays & Vectors', which includes 'Fixed Size Array', 'Arbitrary Size Arrays', and 'Vectors'; 'Strings'; and 'Summary'. At the bottom of the sidebar, it says 'NPTEL MOOCs Programming in C++'. The main content area starts with the text 'Further,' followed by two bullet points:

- `operator=` can be used on strings in place of `strcpy` function in C.
- `operator<=`, `operator<`, `operator>=`, `operator>` operators can be used on strings in place of `strcmp` function in C

At the bottom right of the slide, it says 'Partha Pratim Das' and '11'.

Further, actually does not just end with adding strings or use the addition operator for concatenation of strings, you can do several other operations.

In fact, you do not actually need any of the string dot h functions that you have in the C standard library and achieve their task by using the more natural operators like you can use an assignment in place of doing string copy, you can use the all the comparison operators less than equal to less than greater than equal to greater than in place of using `strcmp`. We know `strcmp` is a relatively complex functions to use because it can take; it takes two strings; two C strings that is card stack pointers and returns you a value which could either be -1 or be 0 or be +1 depending on which string is larger or equal if the strings are equal and so on.

Now, you do not with string type in C++ you do not need to get into any of this. You can just use the comparison operators and compare strings much in the same way you compare integers or floating point numbers. So, this is a very strong features of C++ and particularly for string, this is an extremely convenient way. So, even without getting into deep understanding of C++ you could just start using string and make all your programs smarter and easier to write.

(Refer Slide Time: 18:34)

The slide has a dark blue header bar with the title "Module Summary". On the left, there is a sidebar with a logo at the top, followed by "Module 03", "Partha Pratim Das", "Objectives & Outline", "Arrays & Vectors" (with sub-options "Fixed Size Array", "Arbitrary Size", "Vectors"), "Strings", and "Summary". The main content area contains two bullet points: "Working with variable sized arrays is more flexible with vectors in C++" and "String operations are easier with C++ standard library". At the bottom, it says "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and "12".

In this module we have shown - how we can work with arrays, how vector really makes it easier to make arrays variable sized and how strings operations can be done very easily in C++ using the string type from the standard library.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 06
Sorting and Searching

Welcome to Module 4 of Programming in C++. In this module we are discussing sorting and searching. So, the objective is to look in to the implementation and use of sorting and searching in C and contrast them with C++.

(Refer Slide Time: 00:41)

The slide has a dark blue header with the title "Module Outline". On the left is a sidebar with a logo and navigation links for "Module 04" and "Partha Pratim Das". The main content area lists topics under "Objectives & Outline": "Sorting" (Bubble Sort, Standard Library), "Searching" (Standard Library, STL: algorithm), and "Summary". Below this is a bulleted list of topics:

- Sorting in C and C++
 - Bubble Sort
 - Using Standard Library
- Searching in C and C++
 - Using Standard Library
- **algorithm** Library

At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" with a page number "3".

These are the topics that we are going to discuss.

(Refer Slide Time: 00:44).

The slide is titled "Program 04.01: Bubble Sort". It features a header with the NPTEL logo and the title. On the left, there's a sidebar with navigation links: Module 04, Partha Pratim Das, Objectives & Outline, Sorting Bubble Sort Standard Library, Searching Standard Library, STL: algorithm, and Summary. Below the sidebar is a circular profile picture of Partha Pratim Das.

The main content area is divided into two columns: "C Program" and "C++ Program".

C Program:

```
// FileName:Bubble_Sort.c:  
#include <stdio.h>  
  
int main() {  
    int data[] = {32, 71, 12, 45, 26};  
    int i, step, n = 5, temp;  
  
    for(step = 0; step < n - 1; ++step)  
        for(i = 0; i < n-step-1; ++i) {  
            if(data[i] > data[i+1]) {  
                temp = data[i];  
                data[i] = data[i+1];  
                data[i+1] = temp;  
            }  
        }  
  
    for(i = 0; i < n; ++i)  
        printf("%d ", data[i]);  
  
    return 0;  
}  
12 26 32 45 71
```

C++ Program:

```
// FileName:Bubble_Sort.cpp:  
#include <iostream>  
using namespace std;  
int main() {  
    int data[] = {32, 71, 12, 45, 26};  
    int n = 5, temp;  
  
    for(int step = 0; step < n - 1; ++step)  
        for(int i = 0; i < n-step-1; ++i) {  
            if (data[i] > data[i+1]) {  
                temp = data[i];  
                data[i] = data[i+1];  
                data[i+1] = temp;  
            }  
        }  
  
    for(int i = 0; i < n; ++i)  
        cout << data[i] << " ";  
  
    return 0;  
}  
12 26 32 45 71
```

A note at the bottom states: "Implementation is same in both C and C++ apart from the changes in basic header files, I/O functions explained in Module 02."

At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" with a page number "4".

So to get started, all of you know sorting, and you must have written several programmes in C to sort a set of integers given in an array. So here, we just show one of the most common sorting algorithms known as the bubble sort. You may or may not have used a bubble sort; if you have done some other selection or insertion sort, it is perfectly fine. We are not going to get into the logic of how the sorting is done. All that I want to show is between the two columns; left is the C programme, right is the C++ programme. Both are trying to do bubble sort using exactly the same strategy, the same algorithm; and, you can see the only difference they have between them, is the use of the IO header and the std namespace. So far as the whole algorithm is concerned, the code is concerned, the arrays and comparisons are concerned, it is just identical. So, the first lesson that we take is, any sorting code that is written in C can be identically used in C++.

(Refer Slide Time: 01:57)

The slide is titled "Program 04.02: Using sort from standard library". It features two columns: "C Program (Desc order)" and "C++ Program (Desc order)".

C Program (Desc order):

```
// FileName:qsort.c
#include <stdio.h>
#include <stdlib.h>

// compare Function Pointer
int compare(const void *a, const void *b) {
    return (*(int*)a < *(int*)b);
}

int main () {
    int data[] = {32, 71, 12, 45, 26};

    // Start ptr, # elements, size, func. ptr
    qsort(data, 5, sizeof(int), compare);

    for(int i = 0; i < 5; i++)
        printf ("%d ", data[i]);

    return 0;
}
```

Output: 71 45 32 26 12

- sizeof int, array passed in qsort

C++ Program (Desc order):

```
// FileName:Algorithm_Cust_c++.cpp
#include <iostream>
#include <algorithm>
using namespace std;

// compare Function Pointer
bool compare (int i, int j) {
    return (i > j);
}

int main() {
    int data[] = {32, 71, 12, 45, 26};

    // Start ptr, end ptr, func. ptr
    sort (data, data+5, compare);

    for (int i = 0; i < 5; i++)
        cout << data[i] << " ";

    return 0;
}
```

Output: 71 45 32 26 12

- Size need not be passed.

Module 04
Partha Pratim Das
Objectives & Outline
Sorting
Bubble Sort
Standard Library
Searching
Standard Library
STL:
Algorithm
Summary

NPTEL MOOCs Programming in C++ Partha Pratim Das 5

Now, I would like to illustrate that, if I need to sort a set of numbers or set of strings, do I really every time write a sorting programme myself? I do that when I am trying to learn algorithms or when I am trying to learn the language, but subsequently if I want to sort a set of numbers, I would not like to write the algorithm myself; rather, I will again go back to the standard library and try to use what standard library provides. Now, let us concentrate on the left column, which is the C programme. It is using a sorting function provided in the C standard library, in the header <stdlib.h>. This function is known as Q sort. The name certainly refers to the fact that this function uses quick sort as the underlying sorting algorithm.

Now, please concentrate on how the Q sort function is called. You can see that, there are 4 parameters. So let us see, what are the things that you need to tell the Q sort programme, so that it can sort your data. Certainly, you need to tell where the data exists. So, that is the container, which is an array data here. So, that is the first parameter. Now, the array would have any number of items, any number of elements; how will Q sort, know how many elements does it have or out of the containing elements, how many of them you want to get sorted. So, the second parameter tells Q sort that, how many of the array elements you want to get sorted. It means that, you always start with an index 0 and take as many elements as specified in the second parameter. So, here the second

parameter is 5, which means that Q sort should sort data array from index 0 to index 4, which in this case happens to be the whole array.

The third parameter is little tricky, and right now you may not understand why the third parameter is required. The third parameter says that please provide the size of every element in bytes. Now, this is an integer array, which means that every element is an int. So, the size of every element can be computed from using the sizeof operator in C, so you specify sizeof(int) which for a 32 bit system will pass 4, since int, typically in 32 bit is represented by 4 bytes as the third parameter. May be difficult for you to understand exactly what we need this, but the basic logic is that, since Q sort does not know what type of data your array contains, it could contain int type of elements; it could contain char type of elements; it could contain pointed type of elements; it would contain structure type of elements; it is not possible for Q sort to know after it has got the address of the zeroth indexed element where should it find the first indexed element. So, if it knows the size of the element it will be able to add that as an offset to the starting address of the array to get the first element. Then it can again add that same size of int offset to the first element's position to get the address of the second element and so on. So, that is the implementation for which you need this information. So, that is the third parameter.

The fourth parameter is the most interesting. The fourth parameter is a function pointer. In module 1, you will remember we have talked about function pointers in depth. We have recapped that and if you just move your attention to the top, between the includes and main you find that a function pointer compare has been defined, implemented for the purpose of use by Q sort. Now, what is the role of this compare function? This compare function will take 2 values compare them and say which one the first one is smaller or the second one is smaller. So here, we are shown a compare function, where we are using less than as a comparison. If the first one is smaller than the second one, it will return it true and otherwise it will return a false. This is the basic purpose of the compare function.

Now you will wonder why do I need to do such a complicated thing as writing a separate function, pointer function and passing that function pointer. You will please have to

understand that, when Q sort was written in the library or if you want to write a Q sort kind of function now and you are not told what kind of elements we will need to sort, how will you know how to compare those elements? For example, int can be compared very easily, but if I have say instead of int, if I now give you an array of strings. Now, the array of strings cannot be compared simply by writing a less than or greater than in C. We are not talking about C++ string right now, which you already know has a comparison operator. So, in C if I have to compare strings my values are actually pointed to character, but my comparison has to happen to the use of some strcmp() function. Now how would Q sort know, Q sort did not know what kind of data you will give Q sort to compare them, Q sort also does not know how to compare 2 such data items. And it is your responsibility to provide that compare function to Q sort.

So you need to have this fourth parameter, which is the compare function pointer. If you look into the header, the signature of the compare function pointer you see something which is even more disturbing. You can see that the parameters are declared as of const void * that is there const pointers to constant data of unknown type, why is it that? Because again, Q sort could not have put a signature for the compare function pointer, because it does not know the type of data items. So, all of these assuming that you have a pointer to some type which is not known and, since you do not know the type while you actually want to do the comparison in the return statement of the compare function, you will have to first tell the compiler that, what indeed I now have is an integer. So, you cast the void * pointer first to int pointer, then you de-reference that int pointer to get the integer value and then you compare the two integer values.

So, much need to happen for the Q sort to be used. And that is one of the reasons possibly that even though Q sort is available in the C standard library as a default sorting algorithm, the use of Q sort has not been that popular in the C community. Often, people prefer to write their own functions; bubble sort, selection sort or insertion sort, merge sort, and given the data type of elements that they want to sort and just use that.

Let us now look at the right side of the column. The C++ way of doing it, I will not get into the differences in terms of IO's you already know that. What you must focus in terms of include is, we include another C++ standard library header called algorithm,

which is very very interesting. Algorithm is a header, which contains a whole lot of algorithm codes that you need to often use in your programmes. So, algorithm contains one of the components is the sort algorithm. So, what you call Q sort in C, in C++ this component in algorithm is called sort.

We are again sorting the same data; we are again sorting in the same order. What I need to specify for sort, let us look at the parameters. Certainly, the container which is same as what I did in Q sort. The second I need to tell, how many elements from the beginning, here this we specify in a little bit different way. In Q sort, we just specified the number of elements; here we pass the pointer to the first element, after point of attention. So, we pass ‘data + 5’ which means that we pass the pointer to the index 5 elements, which is just beyond the part of the array that we want to sort. Which in other words, means that you sort 5 elements starting from data? We will look at these details later on, this is called a range in C++.

The third parameter as of Q sort is not required, because as you have seen before also, that C++ compiler can deduce things based on the type. So type, it knows that data is of int type array. So, the elements will be of int types. So, you do not need to separately specify the size of int as a parameter. The third parameter of sort is same as the fourth parameter of Q sort that you need to give the comparison algorithm. This certainly cannot be deduced, because depending on which type of data you are sorting, your comparison strategy would be different. But, what is significant is, if you again move your attention back to where this compare function is defined in the C++ part.

Now you do not have that complication that you had in C, of using parameters which were of const void * type or the expression which does a whole lot of casting and de-referencing. Now, you can declare the parameters exactly as you declare in case of a normal function, int i, int j and in the return you simply compare the, whatever that comparison logic is we will just write that here. It is pretty much like a normal comparison function and that function pointer; function name will be passed as a third parameter. You can do this in C++, because of several features that C++ introduces which we will slowly introduce to you. So, you will understand why this was not

possible in C, but it is possible in C++ to write the compare function with any type that the sort function did not know beforehand, but can still be able to absorb that.

So, with this certainly, it becomes a lot more elegant and efficient to use the sort programme from the standard library. And the consequence therefore is that, the C++ programmers really write their own sorting programmes, they just make use of the sort as is given here. There are small nuances to note; the convention of the direction of sorting, whether it is decreasing order or increasing order differs between the Q sort implementation and the sort implementation. So, if you again focus on the two compare functions that we have written, you will find that in C we have used a less than to be true, in C++ we have given greater than to be true because, in both cases we want the sorting to be done in descending order. So, this is just a matter of convention that exists in C++ sort.

(Refer Slide Time: 14:53)

The slide has a dark blue header with the title "Program 04.03: Using default sort of algorithm". On the left, there's a sidebar with a logo, the text "Module 04", "Partha Pratim Das", "Objectives & Outline", "Sorting", "Bubble Sort", "Standard Library", "Searching", "Standard Library", "STL: algorithm", and "Summary". Below the sidebar is a circular portrait of Partha Pratim Das. The main content area contains a code block labeled "C++ Program" with the following code:

```
// FileName:Algorithm_Cust_c++.cpp
#include <iostream>
#include <algorithm>
using namespace std;

int main () {
    int data[] = {32, 71, 12, 45, 26};

    sort (data, data+5);

    for (int i = 0; i < 5; i++)
        cout << data[i] << " ";

    return 0;
}
```

The output of the program is shown as "12 26 32 45 71". Below the code, there's a bullet point: "Sort using the default sort function of algorithm library which does the sorting in ascending order only." At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

This is just another version of the sort. Here, if you look into the, particularly the call of the sort function, you see that, the first two parameters are there, but the third parameter is not there that is, the compare function is not provided here. This is a short cut that C++ standard library allows you; is if you are trying to sort arrays of type which is known to the C++, that is of the built in types then it is optional to give the comparison function.

The comparison function is not necessary to be provided and if you do not provide that, then by default the sorting happens in the ascending order. If you still want to sort this array in descending order, you will again need to provide the compare function because your direction of comparison, that is whether the first parameter being greater than the second is true or otherwise, will change between the ascending sorting and the descending sorting.

(Refer Slide Time: 16:08)

The slide is titled "Program 04.04: Binary Search". It features a sidebar with navigation links: Module 04, Partha Pratim Das, Objectives & Outline, Sorting, Bubble Sort, Standard Library, Searching, Standard Library, STL, algorithm, and Summary. The main content area is divided into two columns: "C Program" and "C++ Program".

```

C Program
// FileName:Binary_Search.c
#include <stdio.h>
#include <stdlib.h>

// compare Function Pointer
int compare (const void * a, const void * b) {
    if ( *(int*)a < *(int*)b ) return -1;
    if ( *(int*)a == *(int*)b ) return 0;
    if ( *(int*)a > *(int*)b ) return 1;
}

int main () {
    int data[] = {1, 2, 3, 4, 5};
    int key = 3;

    if (bsearch (&key, data, 5,
                sizeof(int), compare))
        cout << "found!\n";
    else
        cout << "not found.\n";

    return 0;
}
found!

```

```

C++ Program
// FileName:Binary_Search_c++.cpp
#include <iostream>
#include <algorithm>
using namespace std;

int main() {
    int data[] = {1, 2, 3, 4, 5};
    int key = 3;

    if (binary_search (data, data+5, key))
        cout << "found!\n";
    else
        cout << "not found.\n";

    return 0;
}
found!

```

At the bottom left is the footer "NPTEL MOOCs Programming in C++" and at the bottom right is "Partha Pratim Das" and the number "7".

Next, we will move on to Binary Search. We all know that, a binary search is a very frequently used algorithm in programming, which given an array of elements, which are already sorted. Binary search can find out whether a given key exists in that array, if it does then it can also say, what is the position of that element in the array? This is, of the simple searching mechanisms, this is one of the most efficient one. And, both C and C++ standard library have mechanisms to do binary search. In C, it again is available from `<stdlib.h>`. It is a function called `bsearch`. In C++, it is again available from component `algorithm` and it is called `binary_underscore search`. If we will look into how this is used in C look on the left hand side, by '`bsearch`' the first parameter is a key. The key has to be passed as its address, so you pass `&key`.

There is a reason why you need to do this, because again, you do not know the type of elements that you have in the array, and therefore the type of the key that you want to search. So, you do not know the type of the variable key in general. So, you cannot write ‘bsearch’ with that. so you again make use of the fact that you can use a pointer to int and consider that to be a pointer to void. So, bsearch uses the first parameter which is void *. And that is the reason you need to give it the address. The second parameter is, the array in which you want to search, certainly this has to be sorted. The third parameter is the number of elements. This is pretty much like Q sort. The fourth parameter is the size of every element and the fifth parameter is the function pointer of the comparison function.

Point to note is, binary search has to make a three way decision, unlike sorting where you just need to know whether it is less or is not less, but binary search it is looking at a particular element in an array, it has to deal with 3 possibilities. One, is the element that it is looking at, may actually be equal to the key, so then you are done with the search. So, you do not need to do anything, you just return that index value of the position of the array. Second, it could be less than that. If you have your array sorted in the increasing order and your value is less than, the element that you are looking at, your key is less than the element that you are looking at, you know that you have to look at the left part of the array. And in the third case, you have to look at the right part of the array. So, your comparison function has to be a three way function, pretty much like the strcmp() function we have in the string.h standard library in C. So on top here, I show how you can write such a compare function for the case that we are dealing with which returns minus 1, 0 or plus 1 as the case may be. And, all the nuances of using parameters as void * pointer and casting them, dereferencing them, as we did it the case of Q sort, will also be involved in this case as well.

Let us look at the C++ form for this. The `binary_search()` function from algorithm component takes the first parameter as the container, which is the data, the second parameter is the end of the range. This is pretty much like sort as we have seen. And it takes the third parameter, which is key. And here, I am showing an example, where the compare function is not explicitly provided, because as I have already discussed in case of sort that, if you are doing a search for elements of built in type the compiler already knows how the comparison is done. So, you do not need to explicitly put a function

pointer for comparison. And, with this, the binary search will happen. If you compare these two you will find, the ease of using the binary search in C++ is immensely more compared to the ease of using it in C. And again, for that matter binary search in C++ is very often used and nobody will write a programme for doing a binary search for any type of data container that someone has.

(Refer Slide Time: 21:24)

The slide is titled "The algorithm Library". On the left, there is a sidebar with a logo at the top, followed by "Module 04", "Partha Pratim Das", "Objectives & Outline", "Sorting", "Bubble Sort", "Standard Library", "Searching", "Standard Library", "STL: algorithm", and "Summary". Below the sidebar is a circular profile picture of Partha Pratim Das. The main content area contains text about the algorithm library and a bulleted list of functions.

The algorithm library of c++ helps us to easily implement commonly used complex functions. We discussed the functions for sort and search. Let us look at some more useful functions.

- Replace element in an array
- Rotates the order of the elements

NPTEL MOOCs Programming in C++ Partha Pratim Das 8

Specifically, in this algorithm library which is a very interesting library, because what it is saying that, if you are using C++, we give you a whole set of common algorithm. So, we just saw how sort and search can be used from algorithm library. There are several others, like replacing elements in an array, rotating the order of elements.

(Refer Slide Time: 21:48).

The slide title is "Program 04.05: replace and rotate functions". It features a table comparing two functions: "Replace" and "Rotate".

Replace	Rotate
<pre>// FileName:Replace.cpp #include <iostream> #include <algorithm> using namespace std; int main() { int data[] = {1, 2, 3, 4, 5}; replace (data, data+5, 3, 2); for(int i = 0; i < 5; ++i) cout << data[i] << " "; return 0; }</pre>	<pre>// FileName:Rotate.cpp #include <iostream> #include <algorithm> using namespace std; int main() { int data[] = {1, 2, 3, 4, 5}; rotate (data, data+2, data+5); for(int i = 0; i < 5; ++i) cout << data[i] << " "; return 0; }</pre>
1 2 2 4 5	3 4 5 1 2
• 3rd element replaced with 2	• Array circular shifted around 3rd element.

NPTEL MOOCs Programming in C++ Partha Pratim Das 9

So, these are examples of replace, rotate codes. We will not go through step by step through them. By now, you should be familiar to use to understand this. Please look up the manual or the book, for the details on these algorithms and start using them. You will really find that, writing things in C++ using the algorithm library becomes very very easy, because most of the common algorithms are already available and very easy to use.

(Refer Slide Time: 22:20).

The slide title is "Module Summary".

The summary points include:

- Flexibility of defining *customised* sort algorithms to be passed as parameter to sort and search functions defined in the algorithm library.
- Predefined optimised versions of these sort and search functions can also be used.
- There are a number of useful functions like rotate, replace, merge, swap, remove etc in algorithm library.

NPTEL MOOCs Programming in C++ Partha Pratim Das 10

In this module, we have shown that how basic sorting and searching can be done in C++ with a much better ease and efficiency and we particularly illustrate that, unlike C where we often end up writing our own sorting code and searching code, in C++ there would be hardly be any reason to do so. No matter what kind of container and data we are using we will show that, this is also true for several other algorithms that we need to use, including merge, swap, remove, all of these, different ones. And, I would encourage you to study these algorithm components and start using them. And then the beauty of the whole thing is, you really do not need to know a lot of C++ to be able to use them, because their use their design and the way they are organised are quite intuitive and you can just study from the manual and start using them.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 07
Stack and Its Applications

Welcome to module 5 of Programming in C++. We have been discussing various example programs that typically you have written in C, and now we are showing how they can be written equivalently in C++. How often the C++ standard library can be used for it and with that, we would like to show, how C++ enhances the ease of programming. So, in this module we will talk about stacks and its applications.

(Refer Slide Time: 00:58)

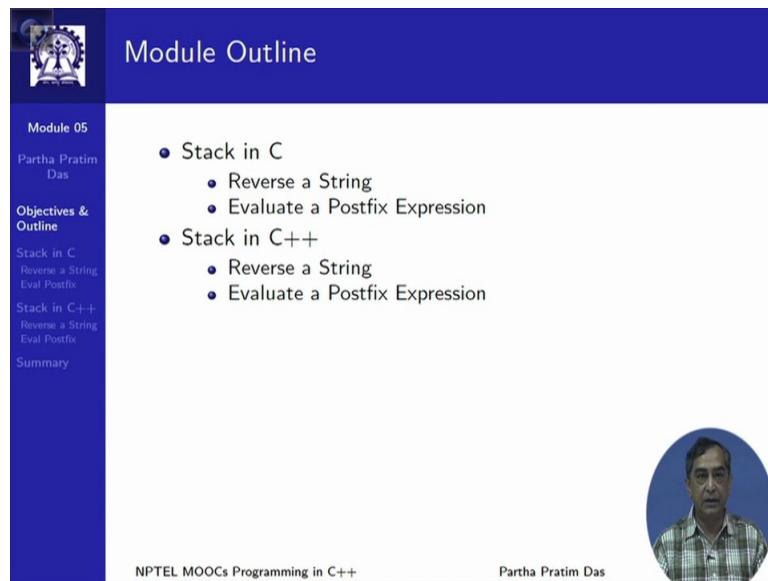
Module Objectives

- Understanding implementation and use of stack in C
- Understanding stack in C++ standard library and its use

NPTEL MOOCs Programming in C++ Partha Pratim Das 2

We will try to understand implementation and use of stack in C, which I presume all of you know and then we will show, how in C++ the standard library can be used for doing stacks.

(Refer Slide Time: 01:10)

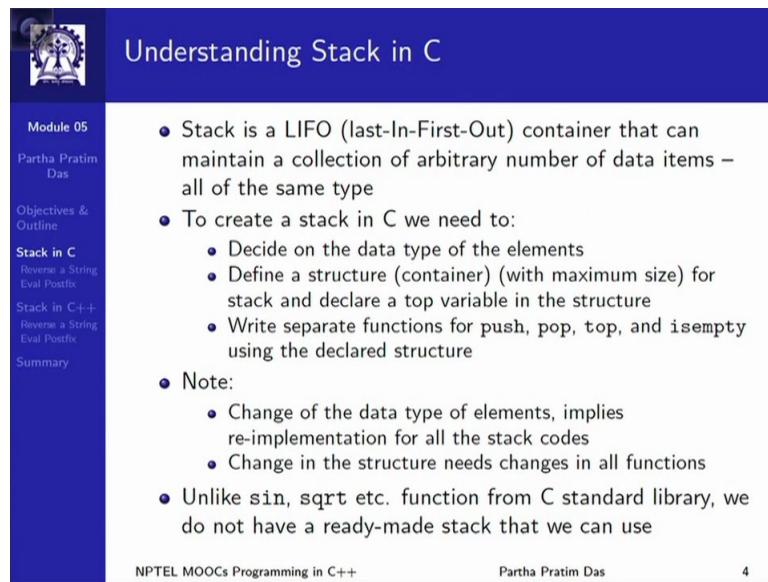


This slide shows the module outline for NPTEL MOOCs Programming in C++. The sidebar on the left lists topics under Module 05, including Objectives & Outline, Stack in C (Reverse a String, Eval Postfix), Stack in C++ (Reverse a String, Eval Postfix), and Summary. The main content area is titled "Module Outline" and contains two sections: "Stack in C" and "Stack in C++". Each section has two bullet points: "Reverse a String" and "Evaluate a Postfix Expression". A circular profile picture of Partha Pratim Das is located on the right side of the slide.

- Stack in C
 - Reverse a String
 - Evaluate a Postfix Expression
- Stack in C++
 - Reverse a String
 - Evaluate a Postfix Expression

So, these are the topics we will primarily use reversing a sting and evaluating a postfix expression as examples.

(Refer Slide Time: 01:22)



This slide is titled "Understanding Stack in C". The sidebar on the left lists topics under Module 05, including Objectives & Outline, Stack in C (Reverse a String, Eval Postfix), Stack in C++ (Reverse a String, Eval Postfix), and Summary. The main content area contains a bulleted list of points about stacks:

- Stack is a LIFO (last-In-First-Out) container that can maintain a collection of arbitrary number of data items – all of the same type
- To create a stack in C we need to:
 - Decide on the data type of the elements
 - Define a structure (container) (with maximum size) for stack and declare a top variable in the structure
 - Write separate functions for push, pop, top, and isempty using the declared structure
- Note:
 - Change of the data type of elements, implies re-implementation for all the stack codes
 - Change in the structure needs changes in all functions
- Unlike sin, sqrt etc. function from C standard library, we do not have a ready-made stack that we can use

Just to recap, on the Introduction of Stack. Stack is a LIFO structure; last-in-first-out container that maintain a collection of arbitrary number of data items. There is no

restriction like in array; there is a restriction of how many elements I can keep depending on the size of the arrays. Stack by definition unbounded, all elements data elements that keep in a stack have to be of the same type, but I can keep any number of elements. So, to create a stack in C, we need to do; these are the common steps that we will need to follow.

First we need to decide on the data type of the element, define a structure or container, for actually defining that in C we will need to use some maximum size, otherwise the C compiler is not allow us to define it, declare a top variable which will maintain the top of the stack and then we will need to write the four functions that are typically required for the operations of that stack.

The function push, which adds an element to the stack; function pop, which removes the top most element, the element last added; function top, which gives up me the top most element without removing it and finally, we will need to have a function isempty() or empty, which will tell me whether there is at all any elements in the stack. So, if isempty() is true then operations like pop and top will be invalid because there is no top element to remove or to return.

So, we can note that as we change the type of elements for a stack, we need to reimplement, rewrite the whole code of the stack that is the code of the stack for stack of integers, and the code of the stack for stack of strings would be different, though the basic notion of the LIFO will be valid in both cases. So, unlike say sine, square root all these functions that we have in C standard library, we have taken at look earlier in math.h, we have a lot of library functions. So, if I want to do compute sine or compute square root or compute arctan, I do not need to write that function I can just use it from the library.

In contrast, C standard library does not give me a mechanism to use a stack. It does not give me a stack which I can use right away. So, this is the current state of affairs with the C programming. So, in this context, we would try to take a look as to how things can change if you use C++.

(Refer Slide Time: 04:15)

The slide has a dark blue header with the title 'Common C programs using stack'. On the left, there's a vertical sidebar with navigation links: 'Module 05', 'Partha Pratim Das', 'Objectives & Outline', 'Stack in C', 'Reverse a String', 'Eval Postfix', 'Stack in C++', 'Reverse a String', 'Eval Postfix', and 'Summary'. The main content area starts with 'Some common C programs that use stack:' followed by a bulleted list. The list includes: 'Reversing a string' (with input ABCDE and output EDCBA), 'Evaluation of postfix expression' (with input 1 2 3 * + 4 - and output 3, showing stack states with values 1, 2, 3, 6, 7, 4, 3), 'Identification of palindromes (w/ and w/o center-marker)', 'Conversion of an infix expression to postfix', and 'Depth-first Search (DFS)'. At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

So, let us also take a look in some of the common problems that we tried to solve using a stack, given a string. I would like to reverse the string that is the left to right order I would like to change to right to left order is a typical program. Another is evaluation of a postfix expression as you all know typically we write expressions with infix notation where the operator happens in between the operands. So, I am showing an example $1 + 2 * 3 - 4$, if I have to evaluate this expression; though we read the expression from left to right, when we go to evaluate this we will not strictly be able to evaluate left to right.

We know that star the multiplication has a higher precedence compared to addition and subtraction. So, first we will have to multiply 2 with 3, get the result 6 and then we have a plus and a minus which both of which has a same precedence, but they are left associative. So, we will have to proceed from left to right and compute that in the result will turn out to be 3.

So, infix notation is very good in terms of how we write an expression. We have been taught from our school days to write these expression and follow the so called BODMAS rules to evaluate it, but it gets very difficult for evaluating a postfix expression because the order of the operators to be executed where is in between depending on their

precedence associativity. So, we take resort to a different notation known as a postfix notation.

In the postfix notation, it is, first the operand has specified and then the operator comes that is why it is postfix. So, always if I have C an operator then the operands for that have already occurred. So, if the operator has an arity 2, then I know that 2 immediately preceding operands are to be applied for that operator. So, we show an example here where you can see that if I look into say this expression which is converted from the infix expression as given here, my expression turns out to be $1\ 2\ 3\ * + 4\ -$ and this also a problem as to how to convert the infix expression to a postfix expression. That also can be solved by using a stack, but here we are just showing that, if the postfix expression is given, how do we evaluate it using a stack.

So, these are the different stacks states all that we can keep on doing is we take; scan the expression from left to right, take a free operands, put it into the stack, keep on stacking till we come across an operator. So, here we push 1, then we push 2, then we push 3 and then we come across the operator star which by definition has arity 2. So, I know that it will require the last 2 operands, we will pop these 2 and 3 apply the operated to get 6 and push it back.

Then we get plus the next operator. So, we know that also needs two operands. So, we pop 6 and 1, the stack becomes empty, we operate it with plus to get 7 and push it back and we proceed in this way to get the result. So, this is a nice example to see that we need to proceed using the different functions of the stack to actually compute the postfix expression that is available to us.

Similarly, several other problems like identification of palindromes, which read the same from both sides, with or without a center marker that is a special character at the middle can be identified by using stack, infix expression can be convert it to postfix depth first search can be done and an examples are several. So, stack plays a significant role in the programming of any software system and therefore, if you can have convenient, reliable, robust mechanisms to use stack we will be strongly benefitted and then this is what C++ gives us as a readymade solution.

(Refer Slide Time: 08:57)

The slide is titled "Program 05.01: Reversing a string". It features a sidebar with navigation links for Module 05, Partha Pratim Das, Objectives & Outline, Stack in C, Reverse a String Eval Postfix, Stack in C++, Reverse a String Eval Postfix, and Summary. The main content area contains C code for reversing a string using a stack. The code includes a stack structure definition, four member functions (empty, top, push, pop), and a main function that demonstrates the reversal of the string "ABCDE". Below the code, the reversed string "EDCBA" is displayed.

```
// FileName: Reverse_String.c
#include <stdio.h>

typedef struct stack {
    char data [100];
    int top;
} stack;

int empty (stack *p) {
    return (p->top == -1);
}

int top (stack *p) {
    return p -> data [p->top];
}

void push (stack *p, char x) {
    p -> data [++(p -> top)] = x;
}

void pop (stack *p) {
    if (!empty(p)) {
        (p->top) = (p->top) -1;
    }
}

void main() {
    stack s;
    s.top = -1;

    char ch, str[10] = "ABCDE";
    int i, len = sizeof(str);

    for(i = 0; i < len; i++) {
        push(&s, str[i]);
    }

    printf ("Reversed String: ");
    while (!empty(&s)){
        printf("%c ", top(&s));
        pop(&s);
    }
}
```

Reversed String: EDCBA

NPTEL MOOCs Programming in C++ Partha Pratim Das 6

So, let us take a look at reversing a string. This is more of a recap for you, this is a C program which is trying to reverse the string as you can see the stack is defined as a structure, which has an array which is the container of the elements and a marker which is a top index which will keep the current index of the top most element, and these are the implementation of the four functions empty, top, bush and pop; empty just checks if the top value is -1; -1 designates that there is no element because are the minimum index value in the array could be 0. So, if it is minus 1 then we designate that the stack is empty.

The top simply returns the element from the top position; push increments the top position and puts the new element which is been given to be added to the stack on to the stack and pop simply decrements the top point at or the top index. So, that the element that was considered to be the topmost is not considered topmost anymore, element just below it is considered to be the topmost and since we cannot do this operation if the stack is already empty, it will be good to check if the stack is empty before we actually perform the pop operation.

Now, this is; we all know this will here using that if you have a string A B C D E here, we can just go in a for loop, add all these characters A B C D and E, one after the other

into the stack and then if we keep on finding out the top element and popping that top element then certainly E has been added last.

So, that will come out first, D has been added just before that, so it will come out next and in result we will get the string that we are showing here, which is E D C B A and we all are familiar with this. So, this is the way if you have to do the reverse string program, then is not only that you have to write this main function or the function which uses the stack to reverse the string, you also need to write the code that is given on the left column that is the scenario in C.

(Refer Slide Time: 11:25)

The slide title is "Program 05.02: Postfix Expression Evaluation". On the left, there is a sidebar with navigation links: Module 05, Partha Pratim Das, Objectives & Outline, Stack in C Reverse a String Eval Postfix, Stack in C++ Reverse a String Eval Postfix, and Summary. At the bottom of the sidebar, it says "Evaluation 3". The main content area contains two columns of C code. The left column is a stack implementation named "PostFix_Evaluation.c" with functions for empty, top, push, and pop. The right column shows the main function for evaluating the postfix expression "1 2 3 * + 4 -".

```

// FileName: PostFix_Evaluation.c
#include<stdio.h>

typedef struct stack {
    char data [100];
    int top;
} stack;

int empty (stack *p) {
    return (p->top == -1);
}

int top (stack *p) {
    return p->data [p->top];
}

void push (stack *p, char x) {
    p->data [++(p->top)] = x;
}

void pop (stack *p) {
    if (!empty(p)) {
        (p->top) = (p->top) -1;
    }
}

void main() {
    stack s;
    s.top = -1;

    // Postfix expression: 1 2 3 * + 4 -
    char postfix[] = {'1','2','3','*','+','4','-'};

    int i, op1, op2;

    for(i = 0; i < 7; i++) {
        char ch = postfix[i];
        if (isdigit(ch)) push(&s, ch-'0');
        else {
            op2 = top(&s); pop(&s);
            op1 = top(&s); pop(&s);
            switch (ch) {
                case '+':push(&s, op1 + op2);break;
                case '-':push(&s, op1 - op2);break;
                case '*':push(&s, op1 * op2);break;
                case '/':push(&s, op1 / op2);break;
            }
        }
    }
    printf("Evaluation %d\n", top(&s));
}

```

NPTEL MOOCs Programming in C++ Partha Pratim Das 7

A similar scenario is given for evaluating postfix expression, again in C left column is identical to the previous example, where the implementation of the stack is given and on the right column we show how we can use this to actually do the postfix evaluation. This is just the code of the algorithm I explained couple of slides back; you can go through it and understand that will actually, will do the job for us. It does the operation of finding if something is an operand which is done at this point, if you just look at particularly focused at this point.

Then you see that we are identifying if some the characters that is given in the expression is operand or not. So, it is considered to be an operand if it is a numeral. So, in this expression we found out if each digit is ch, that is whether it is a digit can be an operand, single digit operand and then we push it otherwise we consider.

We know that this is an operator and the according to what that operator is it finds out and takes the topmost elements, pops them out and then computes the result and pushes it back. The general code of postfix expression would be much more complicated because you will have to also consider the arity of the operator and then do things based on that here we know all four operators, expected operators are binary. So, all those stacks are not here.

(Refer Slide Time: 13:04)

The slide has a dark blue header with the title "Understanding Stack in C++". On the left, there's a sidebar with a logo, the text "Module 05", "Partha Pratim Das", "Objectives & Outline", and a list of topics: "Stack in C", "Reverse a String", "Eval Postfix", "Stack in C++", "Reverse a String", "Eval Postfix", and "Summary". The main content area contains the following list:

- C++ standard library provide a ready-made stack for any type of elements
- To create a stack in C++ we need to:
 - Include the stack header
 - Instantiate a stack with proper element type (like char)
 - Use the functions of the stack objects for stack operations

At the bottom, the footer says "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and the number "8".

Now, let us move on, so what will happen in C++. In C++, the good thing is the C++ standard library provides us with a readymade stack. So, to create that stack or to use that stack all we need to do is to include a new header called the stack header. Instantiate the stacks, which is defined in the stack header and then just start using the functions of that stack object and we will have the stack ready for us.

(Refer Slide Time: 13:37)

The slide is titled "Program 05.03: Reverse a String in C++". It features a sidebar with navigation links for Module 05, Partha Pratim Das, Objectives & Outline, Stack in C, Reverse a String, and Summary. The main content area contains two code snippets side-by-side. The left snippet is in C++ and the right is in C. Both snippets demonstrate using a stack to reverse a string. The C++ code includes #include <iostream>, #include <string.h>, and #include <stack>. The C code includes #include <stdio.h>. Both snippets show pushing characters onto a stack and then popping them to print the reversed string. Below each code snippet is a list of bullet points. The C++ code's list includes: • No codes for creating stack, • No initialization, • Clean interface for stack functions, • Available in library – well-tested. The C code's list includes: • Lot of code for creating stack, • top to be initialized, • Cluttered interface for stack functions, • Implemented by user – error-prone.

Module 05 Partha Pratim Das Objectives & Outline Stack in C Reverse a String Summary	<pre>// FileName: Reverse_String_c++.cpp #include<iostream> #include<string.h> #include<stack> using namespace std; int main() { char str[10] = "ABCDE"; stack<char> s; int i; for(i = 0; i < strlen(str); i++) s.push(str[i]); cout << "Reversed String: "; while (!s.empty()) { cout << s.top(); s.pop(); } return 0; }</pre>	<pre>// FileName: Reverse_String.c int main() { char str[10] = "ABCDE"; stack s; s.top = -1; int i; for(i = 0; i < strlen(str); i++) push(&s, str[i]); printf ("Reversed String: "); while (!empty(&s)){ printf("%c ", top(&s)); pop(&s); } return 0; }</pre>
NPTEL MOOCs Programming in C++	Partha Pratim Das	9

So, here we are showing the reverse string example. Again on the right is the C program of course, the stack code which we have already discussed and not shown here is just the use of the stack in terms of doing the reversing of the string is shown, and an equivalent code is written on the left hand side. Here, please note that in the C++ code, we do not need to write the stack codes instead all that we are trying to do is, we include a specific header. This is the header includes stack, this includes stack header. We have included, which includes the stack definitions and this is how we instantiate stack.

You had seen this notation earlier when we talked about vectors in contrasts to array is you had seen, if I have it have a vector then I could put the element type within the corner bracket similarly, if I have a stack within this angle brackets we are showing what is the type of the element that the stack will be made of and stack objects name will be s.

Now, we look into this, how we use this. Here in C, if you have to push we have to pass two parameters, we have say what is the stack which is the address &s and what is the element that we want to push. Here, we simply do s dot; you had seen this notation also briefly in the context of vector, but we will talk about this lot more, but this is a typical notation; in the notation like what we use in structures, but somewhat differently

interpreted s.push() which says that in stack s, I want to push and what do I want to push? I want push the str, that particular element.

Similarly, if I want to do this here, I want to do check for emptiness I do empty &s. In contrast, here I will do s.empty() as we can see here. So, you can see that, in terms of the code that use the stack the code is very similar and actually as we will understand the code is simpler to write because I do not have unnecessary parameters as in C, where I have to pass the parameter, the pointer to the stack at every point and I can just keep that outside of the function call and just pass the parameters that are needed which in case of push is one parameter; in case of the rest no parameters are required.

Finally, also note in C, while we define the stack; we also needed to initialize the top marker to minus 1. Here, somehow the system takes care of it the compiler takes care of it. So, I will not need to bother about initializing this top marker. So, in brief if we use the stack from the standard library then we do have a lot of advantage of not having to rewrite the stack codes every time we have a new type of data to deal with. These points are highlighted at the below here and instead, we have a well tested readymade stack available for this.

(Refer Slide Time: 17:07)

The slide is titled "Program 05.04: Postfix Evaluation in C++". It features a sidebar with a logo and navigation links for Module 05, including "Objectives & Outline", "Stack in C", "Reverse a String", "Eval Postfix", and "Summary". A video thumbnail of a man is also present. The main content area contains C++ code for postfix evaluation:

```
// FileName:Postfix_Evaluation_c++.cpp
#include <iostream>
#include <stack>
using namespace std;

int main() {
    // Postfix expression: 1 2 3 * + 4 -
    char postfix[] = {'1','2','3','*','+', '4', '-'}, ch;
    stack<int> s;

    for(int i = 0; i < 7; i++) {
        ch = postfix[i];
        if (isdigit(ch)) { s.push(ch-'0'); }
        else {
            int op1 = s.top(); s.pop();
            int op2 = s.top(); s.pop();
            switch(ch) {
                case '*': s.push(op2 * op1); break;
                case '/': s.push(op2 / op1); break;
                case '+': s.push(op2 + op1); break;
                case '-': s.push(op2 - op1); break;
            }
        }
    }
    cout << "\nEvaluation " << s.top();
    return 0;
}
```

At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

I will just proceed, if you look into the next example. This is again similar to what we did in C, but now we are doing it in C++ for the postfix expression evaluation in postfix notation and this code is again very similar to the C code that we had written with the exception of having to include the stack header, define the stack with this notation and then all the stack functions are similarly written in a little bit different format, but the code otherwise is, as it were in C and with the advantage that we do not have to do anything in terms of the C stack code implementation.

(Refer Slide Time: 17:58)

The slide has a blue header bar with the title "Module Summary". On the left, there is a sidebar with the following navigation links:

- Module 05
- Partha Pratim Das
- Objectives & Outline
- Stack in C
Reverse a String
Eval Postfix
- Stack in C++
Reverse a String
Eval Postfix
- Summary

Below the sidebar is a circular portrait of Partha Pratim Das. The main content area contains the following text:

• C++ standard library provides ready-made stack. It works like a data type

• Any type of element can be used for C++ stack

• Similar containers as available in C++ standard library include:

- queue
- deque
- list
- map
- set
- ... and more

At the bottom of the slide, there is footer text: "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" followed by the number "11".

So, in this module what we have shown is, if you are programming in C++ then you get a major advantage in terms of using stack. You do not need to implement the stack code; you do not need to even be concerned about, what should be the maximum size of the container for the stack. The C++ standard library stack automatically takes care of that. It takes care of how to initialize that top and it can be used for any type of element that I want the stack for, and the very interesting and exciting part of the whole C++ programming and standard library is that the story does not stop just with stack.

All common data structures which are frequently required like queue, which is first-in-first-out; deque, which is double ended queue, where you can add and remove at both ends of the sequence. The list; the singly linked list, the map, which is like name value

pair; set, where the ordering is not important, you can just do union intersection in that. All of these data structures are available readymade in the C++ standard library. So, as we start programming in C++ even before we have understood the detailed nuances of the programming language, we would like to frequently take a look at the C++ standard library and write programs making use of all these data structures and make our programming easier and more robust to use.

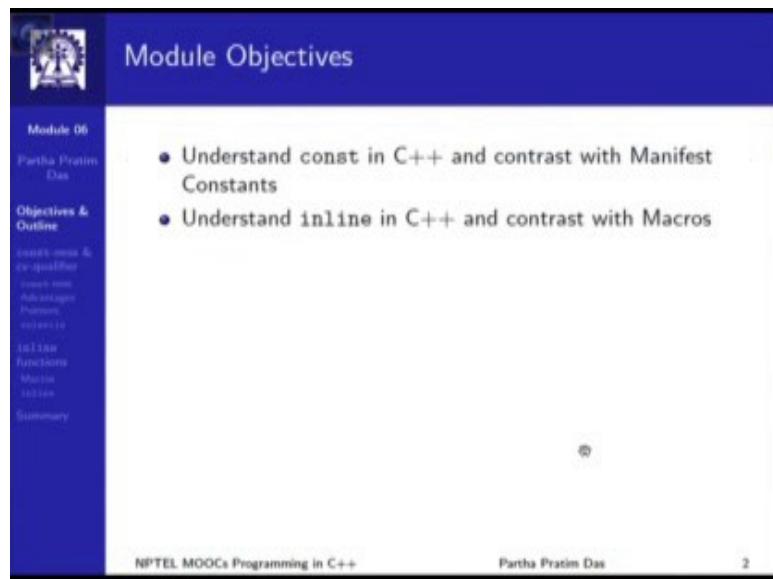
Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 08
Constants and Inline Functions

Welcome to module 6 of Programming in C++. In the first five modules, we have recapitulated the C Programming language, C Standard library and different constructs of the C language itself. We have also taken several examples starting from elementary input output, arithmetic operation, loop kind of examples to use of arrays, strings and specifically data structure, we took example of char to show how programming in C++ and judicious use of the C++ standard library can make programming in C++ really easier more efficient and less error prone.

From, this module onwards we will now start getting into the C++ programming language discussing different features. In the next few modules, we will specifically deal with a set of features which are commonly called as Better C Features that is these features do not make use of the core paradigm of object orientation that exists in C++. But these are procedural extensions to the C language which are required for making the object oriented programming possible and are as such nice features to have they could have been in C also incidentally they were not thought off when C was designed. And the interesting part is that some of this features after they were introduced in C++, and we are going to discuss one of those features right in this module. Some of these features have been later on taken into C programming and are now available in the C 99 standard. We start this module 6, where we will discuss about constants and inline functions.

(Refer slide Time: 02:36)



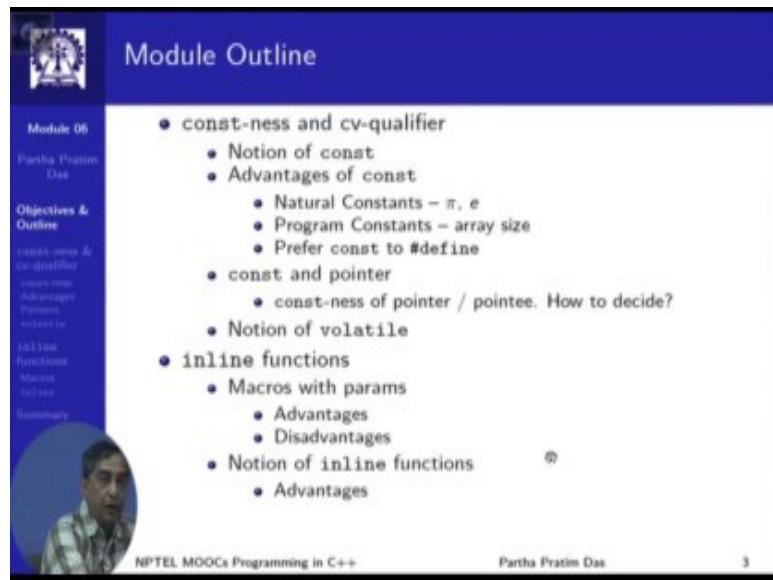
This slide is titled "Module Objectives". It features a sidebar on the left with a logo at the top, followed by "Module 06" and "Partha Pratim Das". Below this is a "Objectives & Outline" section with links to "const-ness & cv-qualifier", "const-ness", "Advantages", "Pointers", "volatile", "inline functions", "Macros", "Summary", and "Summary". The main content area contains two bullet points:

- Understand `const` in C++ and contrast with Manifest Constants
- Understand `inline` in C++ and contrast with Macros

At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" with a page number "2".

So, we will try to understand `const` in C++ and contrast that with the similar concept not exactly the same concept, but similar concept of manifest constant in C and we will try to explain the inline functions in C++ and contrast them with macros.

(Refer slide Time: 03:01)



This slide is titled "Module Outline". It has the same sidebar as the previous slide. The main content area lists the following topics:

- const-ness and cv-qualifier
 - Notion of `const`
 - Advantages of `const`
 - Natural Constants – π , e
 - Program Constants – array size
 - Prefer `const` to `#define`
 - `const` and pointer
 - const-ness of pointer / pointee. How to decide?
 - Notion of `volatile`
 - inline functions
 - Macros with params
 - Advantages
 - Disadvantages
 - Notion of inline functions
 - Advantages

At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" with a page number "3".

So, these are the topics that we will discuss, we will slowly unfold that you can see it on the left side of the screen.

(Refer slide Time: 03:09)

Module 06
Partha Pratim Das

Objectives & Outcomes

const-ness & cv-qualifier

- Course Name
- Advantages
- Pointers
- Memory
- Output
- Summary

Program 06.01: Manifest constants in C

- Manifest constants are defined by #define
- Manifest constants are replaced by CPP (C Pre-Processor)

Source Program	Program after CPP
<pre>#include <iostream> #include <cmath> using namespace std; #define TWO 2 #define PI 4.0*atan(1.0) int main() { int r = 10; double peri = TWO * PI * r; cout << "Perimeter = " << peri << endl; return 0; }</pre>	<pre>// Contents of <iostream> header replaced by CPP // Contents of <cmath> header replaced by CPP using namespace std; // #define of TWO consumed by CPP // #define of PI consumed by CPP int main() { int r = 10; double peri = 2 * 4.0*atan(1.0) * r; // Replaced by CPP cout << "Perimeter = " << peri << endl; return 0; }</pre>
Perimeter = 314.159	Perimeter = 314.159
<ul style="list-style-type: none">TWO is a manifest constantPI is a manifest constantTWO & PI look like variables	<ul style="list-style-type: none">CPP replaces the token TWO by 2CPP replaces the token PI by 4.0+atan(1.0)Compiler sees them as constants

NPTEL MOOCs Programming in C++
Partha Pratim Das
4

So let us start with the Manifest Constants in C. All of us know that we can define a constant value or a fixed value using any literal or using an expression, if we write `#define` followed by a name and then followed by the particular expression that we want to define.

(Refer slide Time: 03:49)

Module 06
Partha Pratim Das

Objectives & Outline
const-ness & cv-qualifier
class new
Access specifiers
Pointers
functions
Matrix
Iteration
Summary

Program 06.01: Manifest constants in C

• Manifest constants are defined by `#define`
• Manifest constants are replaced by CPP (C Pre-Processor)

Source Program	Program after CPP
<pre>#include <iostream> #include <cmath> using namespace std; #define TWO 2 #define PI 4*atan(1.0) int main() { int r = 10; double peri; TWO * PI * r; cout << "Perimeter = " << peri << endl; return 0; }</pre>	<pre>// Contents of <iostream> header replaced by CPP // Contents of <cmath> header replaced by CPP using namespace std; // #define of TWO consumed by CPP // #define of PI consumed by CPP int main() { int r = 10; double peri = 2 * 3.14159 * r; // Replaced by CPP cout << "Perimeter = " << peri << endl; return 0; }</pre>
Perimeter = 314.159	Perimeter = 314.159

CPP replaces the value TWO by 2
CPP replaces the value PI by 3.14159

So, on left we can see examples of this in terms of what we have here; see TWO is defined to be the value 2. Similarly, we have shown how to define pi so which is defined in terms of an expression. PI is; atan(1.0) which is PI by 4, so if you multiply it by 4 you get the value of pi. Then we use them in the expression here to compute the perimeter of a circle. This is a program which is a very commonly used in C and many of you have written this earlier.

Now, let us look into this program little bit differently, let us look on the right hand side. The `#define` that we had here of TWO finally gets replaced at this point by the C preprocessor. So, before the program goes into compilation this line is removed and wherever TWO, this symbol had occurred, earlier as in here will get replaced by wherever I have been defined that symbol to be. So, you can see that pi has been replaced by this whole expression in this and this is the code which actually goes for compilation to the C compiler.

This is the behind the scene scenario and we do not normally unless you put special options in your compiler you would not be able to see this version of the program, where just this `#defines` has been replaced. What is the purpose of the `#define` is to is simply to give the symbol and the expression as equivalent names and C preprocessor can do the

replacement. This is just to make you understand, what is the scenario of a manifest constant?

(Refer slide Time: 06:07)

The slide has a blue header with the title 'Notion of const-ness'. On the left, there's a sidebar with a logo, the text 'Module 06', 'Partha Pratim Das', 'Objectives & Outline', and a list of topics including 'const-ness & re-qualifier', 'CONST-FUN', 'Advantages', 'Disadvantages', 'Exercise', 'alias functions', 'Macro', 'Index', and 'Summary'. Below the sidebar is a circular video player showing a man speaking.

● The value of a `const` variable cannot be changed after definition

```
const int n = 10; // n is an int type variable with value 10
                  // n is a constant
...
n = 5; // Is a compilation error as n cannot be changed
...
int n;
int *p = 0;
p = &n; // Hold n by pointer p
*p = 7; // Change n by p: n is now 7
...
p = &n; // Is a compilation error as n may be changed by *p = 6;
```

● Naturally, a `const` variable *must be initialized when defined*

```
const int n; // Is a compilation error as n must be initialized
```

● A variable of any data type can be declared as `const`

```
typedef struct _Complex {
    double re;
    double im;
} Complex;
const Complex c = {2.3, 7.6}; // c is a Complex type variable
                            // It is initialized with c.re = 2.3 and c.im = 7.6
                            // c is a constant
...
c.re = 3.5; // Is a compilation error as no part of c can be changed
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 5

So, what will be the consequence of this? The consequence is that I wanted to actually use a value which I wanted to treat as constant, but since I have got it replaced if I just again look into this and concentrate on the last line in the comment, I wanted to use that as a constant and in the process the compiler actually never gets to know that they were as a variable called TWO or they were a symbol called TWO, the compiler sees that numerical 3 because it has been replaced. So to take care of this a Notion of Const-ness has been introduced.

(Refer slide Time: 07:02)

The slide is titled "Notion of const-ness". It includes a sidebar with navigation links for Module 06, Partha Pratim Das, Objectives & Outline, const-ness & its qualifier, and a summary. The main content area contains three bullet points:

- The value of a const variable cannot be changed after definition
- Naturally, a const variable must be initialized when defined
- A variable of any data type can be declared as const

Below the bullet points are several code snippets with annotations:

- const int n = 10; // n is an int type variable with value 10
n = 5; // Is a compilation error as n cannot be changed
- int m;
int *p = 0;
p = &m; // Hold m by pointer p
*p = 7; // Change m by p; m is now ?
p = &n; // Is a compilation error as n may be changed by *p = 5;
- const int n; // Is a compilation error as n must be initialized
- typedef struct _Complex {
 double re;
 double im;
} Complex;
const Complex c = {2.5, 7.6}; // c is a Complex type variable
// It is initialized with c.re = 2.5 and c.im = 7.6

A red circle highlights the error message for the assignment to 'n' in the third code snippet.

So, if we look into how const-ness is done, so you see we are doing a declaration where we prefix the declaration of n by a new keyword const. If I just write int n initialize 10 we know n is an integer type of variable whose initial value is 10. We are prefixing it with this const keyword, if we do that what it means is initial value of n is 10 and it also says that it cannot be changed in future, that is by any assignment or by any other means I can change n, n will remain to be 10 all through the program. So, if I try to do something like this here, which is in assigned 5 and try to compile that code, the compiler will give an error will say that n is constant it cannot be changed.

I can try to bypass that and do something like this as in here, as usually if I had another variable m and a pointer p which is an integer type pointer I take the address of m and p I can certainly use the pointer to change the value of m, if I do assign seven to *p what it means that it actually changes m. But, if I try to do the same thing here, if I try to change this value of n by assigning the address of n into the pointed variable p and subsequently, possibly I can do *p assigned 5, I will not be allowed to do that. So, you may be little bit surprised that if we define a variable to be const and then try to use a pointer and take its address even that is given to be a compilation error. And the reason it is a compilation error is if this is not an error then you will be able to do this, which is in the violation of

the principle of const-ness that we are trying to define, that we are trying to say that n cannot be changed.

(Refer slide Time: 09:24)

The slide has a blue header with the title 'Notion of const-ness'. On the left, there is a sidebar with a logo, the text 'Module 06', 'Partha Pratim Das', and a list of topics: 'Objectives & Outline', 'const-new', 'const-qualifier', 'const-new', 'Advantages', 'Disadvantages', 'Example', 'Summary'. The main content area contains three bullet points:

- The value of a const variable cannot be changed after definition
- Naturally, a const variable must be initialized when defined
- A variable of any data type can be declared as const

Below the bullet points is some C code:

```
const int n = 10; // n is an int type variable with value 10
                  // n is a constant
...
n = 5; // Is a compilation error as n cannot be changed
...
int n;
int *p = 0;
p = &n; // Hold n by pointer p
*p = 1; // Change n by p; n is now ?
...
p = &n; // Is a compilation error as n may be changed by *p = 6;
```

At the bottom of the slide, there is a navigation bar with icons for back, forward, search, etc., and the text 'Page: 9/9'.

What is a consequence of that? The next natural consequence of that is a const variable must be initialized. As soon as it is getting defined it must be initialized, because if you do not initialize it then there is no way to change its value so whatever garbage value it has that garbage only will. So, if you declare a const variable without initialization that will become a compilation error

We can also declare variables of different types as const here is an example of using the struct type to variable which is a complex number say and we can define that to be constant which will mean that with this const-ness you will no more be able to change the value of the variable of a component say c.re. c. re by definition has become 2.3 because we have initialized and because we have said that c is const, if c is const then whole of it is const I cannot change any of the component. So, if I try to assign 3.5 to c.re it will be a compilation error. This is the notion of the const-ness.

(Refer slide Time: 10:37)

The screenshot shows a Windows desktop environment with a blue taskbar at the bottom containing icons for various applications like File Explorer, Google Chrome, and Microsoft Word. The main window is titled "Program 06.02: Compare #define and const". It has a dark blue header bar with the title and a "Page 10 / 10" link. Below the header is a navigation bar with a logo, "Module 06", "Partha Pratim Das", and several menu items: Objectives & Outline, Questions & Quizzes, Assignments, Previous, Next, Callouts, Functions, Macros, Index, and Summary.

The main content area is divided into two columns:

Using #define	Using const
<pre>#include <iostream> #include <cmath> using namespace std; #define TWO 2 #define PI 4.0*atan(1.0) int main() { int r = 10; double peri = TWO * PI * r; cout << "Perimeter = " << peri << endl; return 0; }</pre>	<pre>#include <iostream> #include <cmath> using namespace std; const int TWO = 2; const double PI = 4.0*atan(1.0); int main() { int r = 10; double peri = TWO * PI * r; // No replacement by CPP cout << "Perimeter = " << peri << endl; return 0; }</pre>
Perimeter = 314.159	Perimeter = 314.159
<ul style="list-style-type: none">• TWO is a manifest constant• PI is a manifest constant• TWO & PI look like variables• Types of TWO & PI may be indeterminate	
<ul style="list-style-type: none">• TWO is a const variable initialized to 2• PI is a const variable initialized to 4.0*atan(1.0)• TWO & PI are variables• Type of TWO is const int• Type of PI is const double	

So let us see how we use it. So, let us now put two programs side by side on the left, the typical C program which uses #define and on the right we write an equivalent program in C++ which makes use of const-ness to achieve the same purpose. Earlier we were writing #define TWO to the value 2, now we are saying that TWO is a variable of type integer which is initialized with 2, but it is a const variable so you cannot change it. The major consequence of this is when this program on the right hand side, when this program gets true, the C preprocessor certainly said it is no #define, so that symbol TWO will not get replaced at this point.

Similarly, the symbol PI will stay and the compiler will get to see that these are the different variables that exist in the program and the compiler knows that these are constant these cannot be changed. So you can achieve the same purpose that you had in C and you get the added advantage that now the compiler can see all of these and compiler would know what is the type of TWO, compiler would know what is the type of PI or for that matter any variable, any value that you define to be constant using the const keyword.

(Refer slide Time: 12:24)

The screenshot shows a presentation slide titled "Advantages of const". The slide has a blue header bar with the title and a navigation menu on the left. The main content area contains two bullet points and some code examples.

- Natural Constants like π , e , Φ (Golden Ratio) etc. can be compactly defined and used
- Program Constants like number of elements, array size etc. can be defined at one place (at times in a header) and used all over the program

```
const double pi = 4.0*atan(1.0);           // pi = 3.14159
const double e = exp(1.0);                 // e = 2.71828
const double phi = (sqrt(5.0) + 1) / 2.0; // phi = 1.61803

const int TRUE = 1;                        // Truth values
const int FALSE = 0;

const int NULL = 0;                        // null value
```

Note: NULL is a manifest constant in C/C++ set to 0.

```
const int nArraySize = 100;
const int nElements = 10;

int main() {
    int A[nArraySize];
    for (int i = 0; i < nElements; ++i) // Number of elements
        A[i] = i * i;
```

So you get a lot of advantages in terms of using that. There are two major contexts in which you would like to use constant values; one context is, when you deal with different natural constants like PI like e like the golden ratio phi the boolean truth values false value null value and so on. There are several natural constants that occur in the program certainly you can always define them with const with that they will have their value, they will have their type, and they will have their basic property that natural constants naturally, you cannot change the value of PI or you cannot change the value of E, so that property will also be retained.

In addition, another place where we would frequently use constant is where something is constant for my program or something is constant for a particular function. So, for that we will use the second set of definitions like we can have an array size defined to be a constant we can have number of elements defined to be a constant. These are not universal natural constants, but these are constants for my function. If I have done that then the advantage that we get is when we write the program, we can write them in terms of these variables so that later on if we have to change them we can just change the initialization of the constant which is their possibly in the top of the program or in some header file.

There is one added advantage of doing this, if you do #define, the #define has a scope over the whole file. If I #define some value n to a certain specific constant value then wherever I have n in my program that gets replaced by this #define value. But const is a variable declaration so it can be done in any scope I can do it within a function, I can do it within a block within a function and like any variable declaration the variable declaration of const will also remain limited within that scope, so it is possible that I have the same variable n in the same file occurring in two different functions both in both places is this constant but it has different values, you cannot achieve this kind of effect with #define.

(Refer slide Time: 15:07)

● Prefer const over #define	
Using #define	Using const
Manifest Constant	Constant Variable
<ul style="list-style-type: none"> Is not type safe Replaced textually by CPP Cannot be watched in debugger Evaluated as many times as replaced 	<ul style="list-style-type: none"> Has its type Visible to the compiler Can be watched in debugger Evaluated only on initialization

We summarize that we prefer const over #define because it is not safe in terms of type, it is replaced by CPP where as const is not. So, if you are using a debugger you will not be able to see the #define symbols in the debugger with const you will be able to see that. The other side effect is since the #define replaces the expression at every point it needs to be evaluated as many times as it is replaced, where as in case of const it is evaluated only at the initialization point. So, const certainly has a complete advantage over the #define.

(Refer slide Time: 15:46)

The screenshot shows a presentation slide titled "const and Pointers". The slide content includes:

- Module 06
- Partha Pratim Das
- Objectives & Outline
- const-ness & its qualifiers
- Pointers
- Indirection
- dereferencing
- Functions
- Memoization
- Summary

const and Pointers

- const-ness can be used with Pointers in one of the two ways:
 - **Pointer to Constant data** where the pointee (pointed data) cannot be changed
 - **Constant Pointer** where the pointer (address) cannot be changed
- Consider usual pointer-pointee computation (without const):

```
int n = 4;
int n = 5;
int * p = &n; // p points to n. *p is 5
...
n = 6;      // n and *p are 6 now
*p = 7;     // n and *p are 7 now. POINTEE changes
...
p = &m;     // p points to m. *p is 4. POINTER changes
*p = 8;     // n and *p are 8 now. n is 7. POINTEE changes
```

Now, let us see some consequences of defining const particularly, we will look at the const-ness of pointer type data. In a pointer type data we know that we have a pointer and it points to a variable. So the question is, if we talk about const-ness then whose const-ness are we talking about, are you talking about the const-ness of the pointer or the const-ness of the pointed data. Here, we talk about two things pointer to constant data whether data is constant, but the pointer is not or the pointer itself is constant, but the data may or may not be constant.

Below here I just show a typical example of how we compute with pointer and pointee we have defined two variables, so we have a pointer which takes the address of one variable and then using that I can directly change the variable or I could I can change it through the pointer. Similarly, at this line, earlier the pointer was pointing to n, now it has been changed to point to m and they can again use it to change the value of m. This is the typical use of a pointer-pointee scenario.

(Refer slide Time: 17:23)

The slide has a blue header with the title 'const and Pointers: Pointer to Constant data'. On the left, there's a sidebar with navigation links: Module 06, Partha Pratim Das, Objectives & Outline, const-ness & cv-qualifier, const-one, Advanced Pointers, std::vector, inline functions, Return types, and Summary. The main content area contains three code snippets under the heading 'Consider pointed data':

```
int n = 4;
const int n = 5;
const int * p = &n;
...
n = 6; // Error: n is constant and cannot be changed
*p = 7; // Error: p points to a constant data (n) that cannot be changed
p = &m; // Okay
*p = 8; // Okay
```

Following this, there's an 'Interestingly,' section:

```
int n = 5;
const int * p = &n;
...
n = 6; // Okay
*p = 6; // Error: p points to a 'constant' data (n) that cannot be changed
```

Finally, another section:

```
const int n = 5;
int * p = &n; // Error: If this were allowed, we would be able to change constant n
...
n = 6; // Error: n is constant and cannot be changed
*p = 6; // Would have been okay, if declaration of p were valid
```

At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

So, with this we would next like to discuss as to how we can control these changes with the use of const.

(Refer slide Time: 17:34)

This screenshot is identical to the previous one, but with several lines of code highlighted by red arrows pointing to them from the left. The highlighted lines are:

- const int n = 5;
- const int * p = &n;
- n = 6; // Error: n is constant and cannot be changed
- *p = 7; // Error: p points to a constant data (n) that cannot be changed
- p = &m; // Okay
- *p = 8; // Okay

The rest of the slide content, including the 'Interestingly,' section and the final code snippet, remains the same.

So first is, if I have a pointer to a constant data. So, what we are doing here, is we have written const before the data, before the type of the value that the pointer points to. If I

write the const at this point it means that the pointed data is constant, it cannot be changed. So, n has been defined to be a constant value. We already know an attempt to change that value of n is an error because if n is constant, and we have defined p to be a pointer to n. So, trying to change the value of n using p that is *p assigned 7 is also an error. But, p itself is not a constant, that is if I want now I can make p point to some other variable, so m is a variable here which is not a constant variable. I can make p point to m and then I can use this *p assigned 8 to change the value of m, m will now become 8, it was 4, it will now become 8.

(Refer slide Time: 18:55)

The slide is titled "const and Pointers: Pointer to Constant data". It features a sidebar with a logo and navigation links for "Module 06", "Partha Pratim Das", "Objectives & Outline", "const-const & cv-qualifier", "const const", "Algebraic Pointers", "introduce", "Julian functions", "Matrix class", and "Summary". A video thumbnail of the speaker is on the left.

Consider pointed data

```
int n = 4;
const int n = 6;
const int * p = &n;
...
n = 6; // Error: n is constant and cannot be changed
*p = 7; // Error: p points to a constant data (n) that cannot be changed
p = &m; // Okay
*p = 8; // Okay
```

Interestingly,

```
int n = 5;
const int * p = &n;
...
n = 6; // Okay
*p = 6; // Error: p points to a "constant" data (n) that cannot be changed
```

Finally,

```
const int n = 6;
int * p = &n; // Error: If this were allowed, we would be able to change constant n
```

Handwritten notes on the slide include:
 - A diagram showing a pointer p pointing to a variable n, with a red box around n and the text "n can't be changed".
 - A note below the first code block: "+ Knowledge is used".
 - A note below the second code block: "n = 6; // Okay".

So, now if you will look into, if I have a variable which is not a constant say int has been defined to be an integer type variable initialized with 5 and I have a pointer p which points to a constant type of integer value and I put the address of n into p. Now, naturally m assignment of 6 to n is valid because n itself is not a constant. It is also that *p assigned 6 is valid, but if I try to do *p assigned 6 that is not valid because p says that m pointing to a constant integer. So very interesting scenario ; I have a p here which is pointing to n. P knows, this knows that if I write *p, is constant. That is p cannot be used to change this value, but n by itself is not constant. So, n can be changed.

Now, this is valid because what you are saying is you are saying more than what is required, you are saying that n by itself can change. So, whether I change it directly as n or I change it through a pointer it does not make a difference because n can change, but the pointer as said that I am restricted not to change. The pointer is said that if you go through me then I will not allow you to change the value. So, here this is a scenario where the variable actually can change, but the pointer gives a view which does not allow you to change that, but if I directly go or if I use some other pointer which does not point a constant value then we will be able to change that. Finally, if we try to do the reverse that is if I have a constant variable n and if I try to use a pointer to a non constant value p then however I will not be able to do this.

(Refer slide Time: 21:46)

const and Pointers: Pointer to Constant data

Consider pointed data

```
int n = 4;
const int n = 5;
const int * p = &n;
...
n = 6; // Error: n is constant and cannot be changed
*p = 7; // Error: p points to a constant data (n) that cannot be changed
p = &m; // Okay
*p = 8; // Okay
```

Interestingly,

```
int n = 5;
const int * p = &n;
...
n = 6; // Okay
*p = 6; // Error: p points to a 'constant' data (n) that cannot be changed
```

Finally,

```
const int n = 5;
int * p = &n; // Error: If this were allowed, we would be able to change constant n
```

Handwritten notes:

- A red box encloses the first code snippet with the note "n (Const)" written next to it.
- A red arrow points from the word "Interestingly" to the second code snippet.
- A red box encloses the third code snippet with the note "*p is not const" written next to it.

So, if we just try to illustrate then the last int case here, so we are talking about this case if I have a p which points to n, where this is constant and *p is not constant then we have an error quite validly, because n is a constant. If *p is not constant, *p is trying to point to n then I can always make use of *p here to change the value of n which am not supposed to do.

So what we learned here is a basic notion that if a value is a not a constant I can still use it pointed to a constant to view that, get that, but I will not be able to change it to that

pointer. But if a value is constant then I cannot use a pointer which is a pointed to a non constant value I will not even be allowed to initialize that const pointer with the address of this constant variable because that would violate the basic principle of const-ness.

(Refer slide Time: 23:20)

const and Pointers: *Constant Pointer*

Module 06
Partha Pratim Das
Objectives & Outcome
const-ness & non-const-ness
Pointers
functions
Summary

Consider pointer

```
int n = 4, n = 5;
int * const p = &n;
...
n = 6; // Okay
*p = 7; // Okay. Both n and *p are 7 now
...
p = &m; // Error: p is a constant pointer and cannot be changed
```

By extension, both can be const

```
const int n = 4;
const int n = 5;
const int * const p = &n;
...
n = 6; // Error: n is constant and cannot be changed
*p = 7; // Error: p points to a 'constant' data (n) that cannot be changed
...
p = &m; // Error: p is a constant pointer and cannot be changed
```

Finally, to decide on const-ness, draw a mental line through *

```
int n = 5;
int * p = &n; // non-const-Pointer to non-const-Pointee
...
int m = 6;
int * p = &m; // non-const-Pointer to const-Pointee
```

Next, let us look at the const-ness of the other side, what if the pointer is constant? So, if you look in here this is where we are, we have slightly shifted the position where we had written the const. Earlier the const was written here at this point, now the const is written after the star symbol this says that the pointer is constant, but the value that it is pointing to is not constant. If I draw it p is const n not const.

So, what it means that if I can easily write this n is not const, so I can change its value, since n is not const I can use p dereference it assign seven to *p that would change the value of n which is valid because I am not violating anything, but I will not be able to do is the last one that is I cannot change the address that is stored in p. I cannot make p now point to a new variable m, because I have said that the pointer itself is constant this side is constant now that is earlier the other side was constant. Naturally, if we have this then by extension we can also combine both of this that both the pointer and the data it is pointing to can be constant.

(Refer slide Time: 24:55)

The screenshot shows a presentation slide with a blue header containing the title 'const and Pointers: Constant Pointer'. The slide content is as follows:

Consider pointer

```
int n = 4, *p = &n;
n = 6; // Okay
*p = 7; // Okay. Both n and *p are 7 now
p = &m; // Error: p is a constant pointer and cannot be changed
```

By extension, both can be const

```
const int n = 4;
const int *p = &n;
const int *const p = &n;

n = 6; // Error: n is constant and cannot be changed
*p = 7; // Error: p points to a 'constant' data (n) that cannot be changed
p = &m; // Error: p is a constant pointer and cannot be changed
```

Finally, to decide on const-ness, draw a mental line through *

```
int n = 5;
int *p = &n;           // non-const-Pointer to non-const-Pointee
```

The slide has a sidebar with navigation links: Module 06, Partha Pratim Das, Objectives & Outline, const-mem & -qualifier, Pointers, References, Functions, Metrics, Summary.

So, we here we are showing an example where I write const on both sides which means that p is a constant pointer to a constant data, which means neither p can be made to point to any other variable other than n nor I can use p to change the value of n. So, all of this will now become error.

Now, at the end certainly since we are writing the const-ness on the pointed data or the pointer itself it is confusing at times as to where should I write the const and what will become const by putting the const keyword. The thumb rule is very simple, that when you have this declaration look at the star symbol in the whole declaration.

(Refer slide Time: 26:09)

The slide is titled "const and Pointers: Constant Pointer". It features a sidebar with navigation links for "Module 06", "Partha Pratim Das", "Objectives & Outline", and several sub-sections under "const-ness & its qualifiers".

Consider pointer

```
int n = 4, m = 5;
int * const p = &n;
...
n = 6; // Okay
*p = 7; // Okay. Both n and *p are 7 now
...
p = &m; // Error: p is a constant pointer and cannot be changed
```

Handwritten notes above the code:
((const) int*) p = ...
data |
ptr

By extension, both can be const

```
const int n = 4;
const int m = 5;
const int * const p = &n;
...
n = 6; // Error: n is constant and cannot be changed
*p = 7; // Error: p points to a 'constant' data (n) that cannot be changed
...
p = &m; // Error: p is a constant pointer and cannot be changed
```

Handwritten notes above the code:
int * const p = ...

Finally, to decide on const-ness, draw a mental line through *

```
int n = 5;
int * p = &n;           // non-const-Pointer to non-const-Pointer
```

Handwritten note below the code:
non-const-Pointer to non-const-Pointer

Mentally draw a vertical line so if you are trying to do this you are saying `const int *p` etcetera, draw a vertical line through this star symbol and see which side the `const` keyword appears, this is your data side and this is your pointed side. So if the `const-ness` is on the data side then whatever you are pointing to is constant. In contrast if you have `int * const p` etcetera, so this `const` is on the pointer side so the pointer is constant. So that is the basic thumb rule by which you can decide which of them is a constant.

(Refer slide Time: 26:57)

The slide has a blue header with the title 'const and Pointers: The case of C-string'. On the left, there's a sidebar with navigation links: Module 06, Partha Pratim Das, Objectives & Outline, const-ness & qualifiers, const-ness, Advantages, Pointers, pointers, inline functions, Macros, macros, Summary. The main content area contains code examples and their outputs.

Consider the example:

```
char * str = strdup("IIT, Kharagpur");
str[0] = 'J'; // Edit the name
cout << str << endl;
str = strdup("JIT, Kharagpur"); // Change the name
cout << str << endl;
```

Output is:

```
IIT, Kharagpur
JIT, Kharagpur
```

To stop editing the name:

```
const char * str = strdup("IIT, Kharagpur");
str[0] = 'J'; // Error: Cannot Edit the name
str = strdup("JIT, Kharagpur"); // Change the name
```

To stop changing the name:

```
char * const str = strdup("IIT, Kharagpur");
str[0] = 'J'; // Edit the name
str = strdup("JIT, Kharagpur"); // Error: Cannot Change the name
```

To stop both:

```
const char * const str = strdup("IIT, Kharagpur");
str[0] = 'J'; // Error: Cannot Edit the name
str = strdup("JIT, Kharagpur"); // Error: Cannot Change the name
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 12

The examples are given below. So, you can use this and for string this is an example that I have worked out you could read it carefully and try to understand a string is given and if we just have a string then you can two ways change that either you can edit the string or you change the string itself. On the code on top we show the effect of editing the string or changing the whole string altogether.

(Refer slide Time: 27:34)

The slide has a blue header with the title 'const and Pointers: The case of C-string'. On the left, there's a sidebar with navigation links: Module 06, Partha Pratim Das, Objectives & Outline, const-ness & qualifiers, const-ness, Advantages, Pointers, pointers, inline functions, Macros, macros, Summary. The main content area contains code examples and their outputs, with some parts underlined in red.

Consider the example:

```
char * str = strdup("IIT, Kharagpur");
str[0] = 'J'; // Edit the name
cout << str << endl;
str = strdup("JIT, Kharagpur"); // Change the name
cout << str << endl;
```

Output is:

```
IIT, Kharagpur
JIT, Kharagpur
```

To stop editing the name:

```
const char * str = strdup("IIT, Kharagpur");
str[0] = 'J'; // Error: Cannot Edit the name
str = strdup("JIT, Kharagpur"); // Change the name
```

To stop changing the name:

```
char * const str = strdup("IIT, Kharagpur");
str[0] = 'J'; // Edit the name
str = strdup("JIT, Kharagpur"); // Error: Cannot Change the name
```

To stop both:

```
const char * const str = strdup("IIT, Kharagpur");
str[0] = 'J'; // Error: Cannot Edit the name
str = strdup("JIT, Kharagpur"); // Error: Cannot Change the name
```

Page: 19 / 19

But you can stop that if you do something like, put a const here, if you put a const here then the string itself becomes constant, so you cannot change any character of the string. So, you cannot like in here you could write assign n to the first symbol you cannot do that anymore. Whereas, if you put the const on this side then you can now change every any of the symbols in the string, but you cannot change the string as a whole.

Here you could change the string now you cannot change the string because that means changing the pointer. And certainly you could protect both the edit as well as the changing of the name if you put const on both sides of the pointer that is, if you have a constant char star pointer pointing to a constant array of characters then neither it can be edited nor it can be changed. This is an example to show how const-ness applies on both sides. So, we have discussed the basic notion of const-ness and illustrated how const-ness applies in terms of the pointers.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 09
Constants and Inline Functions (Contd.)

Welcome to module 6 in a programming in C++ again. We have already discussed the notion const-ness in this module, we have discussed how in place of using manifest constant we can use the const declaration to qualify a variable declaration and how it does not allow us to change the value of a variable once it has been declare and initialized. We have also shown how const-ness work with pointers and talked about the constant pointer ans const-ness of the pointed data. We will continue on that.

(Refer Slide Time: 01:02)

The slide has a blue header bar with the title 'Notion of volatile'. On the left, there is a vertical sidebar with a logo at the top, followed by a navigation menu:

- Module 06
- Partha Pratim Das
- Objectives & Outcome
- const-ness & re-qualifier
- new new
- Assignment Operator
- Volatile
- inline functions
- Macros
- Summary

The main content area contains a bulleted list:

- Variable Read-Write
 - The value of a variable can be read and / or assigned at any point of time
 - The value assigned to a variable does not change till a next assignment is made (value is persistent)
- const
 - The value of a const variable can be set only at initialization – cannot be changed afterwards
- volatile
 - *In contrast*, the value of a volatile variable may be different every time it is read – even if no assignment has been made to it
 - A variable is taken as volatile if it can be changed by hardware, the kernel, another thread etc.
- cv-qualifier: A declaration may be prefixed with a qualifier – const or volatile

At the bottom of the slide, there is footer text: 'NPTEL MOOCs Programming in C++', 'Partha Pratim Das', and '13'.

Next, we will talk about a related notion which is known as the Volatile. This volatile is a less commonly known concept so let us try to understand it little bit. Let us think about a variable, so what can we do with a variable, after variable has been declared defined possibly has been initialized then we can read the value of the variable or we can assign a new value at any point of time. The basic property that we always program with is, if I have read the value of the variable then if I read the value again I am expected to get the

earlier value itself unless some new value has been assigned in between. So if I assign a value and keep on reading a number of times I will always get the same value till I make the next assignment, these are the basic concept of read write of a variable.

Now, look at const-ness in this context. In const-ness what we are saying that we are allowed to assign or in that sense initialized the value only once and not allowed to change it afterwards. So, for the whole lifetime of the program the value that I read of this variable will be the same that is the const-ness. Volatile in contrast is saying that it is a volatile value which means that if I read a variable at different points of time there is no guarantee that I will get the same value, I may get different values.

So, even when I have not done any assignment to it so as a volatile refers volatile is something that evaporates. Suppose, the value of the variable was 5 you read it once then you need not make an assignment, but you read it after may be ten statements or after it certain time the value may be found to be 7. Earlier value 5 has disappeared, has evaporated. So that is the notion of the volatile variable.

Why is this important to have this kind of a variable behavior in program because there are some system situations where this can happen, for example, suppose I am writing a network programming code whose purpose is to a keep listening to a port to find, if some data has arrived. So what are you doing you are expected to write anything to that port because you are expecting data from the outside the system, so you are only reading, reading, reading. What will happen? When some data arrives then your value will change, but earlier when you have read the value was possibly null, then suddenly you read it after may be 100 millisecond you find that some value has come, you read it after another 100 millisecond then there may be at different value that has come because another different packet has come.

It is possible that a variable can be changed by the hardware by the kernel of the operating system by another thread and so on, so it is required to module that behavior and in C++ this is module in the conjunction with the const concept because one which kind gives you absolute truth and the other which gives you kind of no guarantee that you just do not know what are the value of the variable is. So they are club together and

we called then as CV qualifier, C for const V for volatile, CV qualifier and any declaration of a variable can be prefix with a CV qualifier const or volatile.

(Refer Slide Time: 05:01)

The slide has a blue header with the title 'Using volatile'. On the left, there's a sidebar with a logo, the text 'Module 06', 'Partha Pratim Das', 'Objectives & Outcome', and a list of topics including 'const, const & no-qualifier', 'array now', 'Advantages', 'Disadvantages', 'volatile', 'inline functions', 'Macros', and 'Summary'. Below the sidebar is a circular profile picture of Partha Pratim Das. The main content area starts with 'Consider:' followed by a code snippet:

```
static int i;
void fun(void) {
    i = 0;
    while (i != 100);
}
```

Then it says 'This is an infinite loop! Hence the compiler should optimize as:' followed by another code snippet:

```
static int i;
void fun(void) {
    i = 0;
    while (1);      // Compiler optimizes
}
```

Next, it says 'Now qualify i as volatile:' followed by a third code snippet:

```
static volatile int i;
void fun(void) {
    i = 0;
    while (i != 100); // Compiler does not optimize
}
```

Finally, it states 'Being volatile, i can be changed by hardware anytime. It waits till the value becomes 100 (possibly some hardware writes to a port).'

At the bottom, it shows 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

We will show example of that. Here, we are trying to show how volatile could be useful. This a code very simple code which assigns the value 0 to i and then tries to look on the condition that i is not equal to 100. Now, if you just had shown this code you will immediately identify that this is an infinite loop. Why it is an infinite loop? Because if the value of i is a 0 and then I am checking if i is 100 and i will continue as long as i is not hundred suddenly the value of i will never become 100. So this condition will always remain true so this loop will continue in definitely, so the what the compiler will do compiler will optimize and say that this is true, this is while(1) simply that whole expression will go away.

Now let us say you qualify this by as a volatile. You say this is a volatile variable i and you read the same code now it does neither gets optimize, and what you expect is this code will actually work because being a volatile variable you expect that there is some other agent, there some hardware, some port, some kernel system, some thread possibly is changing its through some other means. So, you keep on waiting till its value becomes 100 and it is possible that at some point it become 100 and then that a particular

condition becomes false and the function will be able to return. This is the use of the volatile variable in C, C++ as well.

(Refer Slide Time: 06:43)

Program 06.03: Macros with Parameters

Module 06
Partha Pratim Das

Objectives & Outcome
conceptual & re-qualifier
macro
functions
Macros
etc.
Summary

Source Program	Program after CPP
<pre>#include <iostream> using namespace std; #define SQUARE(x) x * x int main() { int a = 3, b; b = SQUARE(a); cout << "Square = " << b << endl; return 0; }</pre>	<pre>// Contents of <iostream> header replaced by CPP using namespace std; // #define of SQUARE(x) consumed by CPP int main() { int a = 3, b; b = a * a; // Replaced by CPP cout << "Square = " << b << endl; return 0; }</pre>
Square = 9	Square = 9
<ul style="list-style-type: none"> • <code>SQUARE(x)</code> is a macro with one parameter • <code>SQUARE(x)</code> looks like a function 	

NPTEL MOOCs Programming in C++ Partha Pratim Das 15

Let us now move on and talk about a different kind of use for `#define` which is again processed by the cp processor we call them macros. The difference being that we still define them with a `#define` word we have a name, but the main thing is we now have a parameter into this. So what happens is when I use, I use it with parameters so we are saying that a square is the defined name and I am putting a parameter to it the effect of that is, the C preprocessor will directly go, match a with x and replace all Xs in the define expression by a. It is simple x base substitution. So, wherever it will find x in that expression it will substitute it with the parameter a.

Now, macros are very commonly used features in C and are also useful in C++, but it kind of allows us to write a function like notation. So if you are not told to `#define` line and if you just reading the main you will not know where the square is a macro or it is a function it could pretty well be a function. But when it comes to the compiler, again like in the case of manifest constant the compiler was not been able to see that it is a variable it was just seeing the constant value that you have written or the constant expression that you have written, here again the compiler will not be able to see any kind of

a function notation he will simply be able to see the expression that the CPP has replaced.

(Refer Slide Time: 08:52)

The slide has a blue header with the title 'Pitfalls of macros'. On the left, there's a sidebar with a logo and navigation links for 'Module 06' (Partha Pratim Das), 'Objectives & Outline', 'Scope-area & co-qualifier', 'Loops now', 'Advantages', 'Functions', 'Macros', 'Exercise', and 'Summary'. The main content area starts with 'Consider the example:' followed by a C++ code snippet. The code defines a macro #define SQUARE(x) x * x and uses it in a main() function to calculate b = SQUARE(a + 1). The output is shown as 7 instead of 16. It then says 'To fix:' and shows the corrected code where the macro is defined as #define SQUARE(x) (x) * (x). The slide footer includes 'NPTEL MOOCs Programming in C++', 'Partha Pratim Das', and the number '16'.

So Macros have been very often widely used in a C programming and it has advantages particularly in C you could not have done without macros for several reasons. It gives you efficiency also because macros do not need a function called over it, but they have a very serious pitfalls also. we show some of the pitfalls here in the same square example and we are trying to, this is a macro and here I am just trying to use it with a +1. Earlier, I was using it with a I am trying to use it a +1, certainly if somebody reads it the mental notion would be, 1 will be added to a so it will become 4 and then it will be squared, so it will become 16.

But, when you run this program we will actually get an output 7 and to understand that you will have to expand the macro, now expand this macro to this line. You can see that in the macro x is replaced by a+1, so in place of x if you just write a+ 1 it becomes a + 1 * a +1 and then it becomes an expression were the precedence and associativity apply. So, 1 into 1 times 'a' which is in the middle, gets the precedence so that is gets a operated first, so this practically becomes 2a + 1. So it becomes 7. So, certainly this cause all the problem.

Fortunately, this problem which can be fixed, if you just put parenthesis around every x or around every instance of the parameter in the macro definition, if you do that, how it will help? We can just see the expansion, this will help, since the parenthesis have been put, after the macro is expanded, in the last line of this slide you can see that there is parenthesis around $a + 1$. So, it says $(a + 1) * (a + 1)$ now the BODMAS rule say that $a+1$ has to happen first which is what we had expected, so this can still be fixed.

Let us go ahead. this is really disturbing that if you have to remember that every time you write x, you have to put parenthesis around that otherwise you might get surprise. Next, let us see an example where the situation is actually worse because, here now we have fixed the definition of the macro now we are trying to use it again and I want to use it with $++ a$, a plus plus is a pre increment. If a is 3, as in here if do plus plus I expected to become 4, 4 should go to the square it should get squared it should become 4 times 4 it should result should be 16. You try that the result is 25. Why?

Again look into the expansion of this macro, if it is expanded it looks like this, because there are two x so $++ a$, is written for each one of them. So what happens, $++$ has the highest precedence over is a higher precedence than multiplication, so both plus plus has happen before multiplication has happened so, a was 3 it first becomes 4 then it becomes 5 and then the multiplication happens. So the result is expectedly 25. The unfortunate part of the story is there is no easy fix for this in C. So, you will have to live with these kinds of possible pitfalls in the macros.

(Refer Slide Time: 12:30)

The screenshot shows a presentation slide with a blue header bar containing the title 'inline Function'. The main content area contains a bulleted list of three points:

- An **inline** function is just another functions
- The function prototype is preceded by the keyword **inline**
- An **inline** function is expanded (inlined) at the site of its call and the overhead of passing parameters between caller and callee (or called) functions is avoided

At the bottom left, there is a footer bar with the text 'NPTEL MOOCs Programming in C++'. At the bottom right, it says 'Partha Pratim Das' and '18'.

So, there is a new feature in C++ which is called as Inline Function. Let me first define how to do this and then we will explain, how does it relate to the macros? An Inline function is just another function is no special kind of function only difference is, in the prototype of the function in the header you write the keyword inline before the return type. So If you write this keyword inline then what happens is when the function is called the actual function call does not happen, but whatever code the function has that code the compiler puts at the call site that is the basic. So, the over rate of function is call is avoided,

(Refer Slide Time: 13:11)

Module 06
Partha Pratim Das

Objectives & Outline
cout-mta & cv-qualifier
class-new
Advantages
Disadvantages
inline functions
Macros
inlining
Summary

Program 06.04: Macros as inline Functions

Using macro	Using inline
<pre>#include <iostream> using namespace std; #define SQUARE(x) x * x int main() { int a = 3, b; b = SQUARE(a); cout << "Square = " << b << endl; return 0; }</pre>	<pre>#include <iostream> using namespace std; inline int SQUARE(int x) { return x * x; } int main() { int a = 3, b; b = SQUARE(a); cout << "Square = " << b << endl; return 0; }</pre>
Square = 9	Square = 9
<ul style="list-style-type: none">SQUARE(x) is a macro with one paramMacro SQUARE(x) is efficientSQUARE(a + 1) failsSQUARE(**a) failsSQUARE(**a) does not check type	<ul style="list-style-type: none">SQUARE(x) is a function with one paraminline SQUARE(x) is equally efficientSQUARE(a + 1) worksSQUARE(**a) worksSQUARE(**a) checks type

NPTEL MOOCs Programming in C++ Partha Pratim Das 19

So, we define the function, we prefix the function header with inline it just focus here on the right hand side which is a C++ code left hand side is the original C code of your reference on the right hand side we have not doing a hash define macro we are saying square is the function which takes x returns x times x as an integer and we are prefixing it with the inline keyword in front. The use code for this here and here remains same.

Now, the advantage is, in C++ this is truly a function. So you cannot have any of the pitfalls that we were showing earlier very truly first evaluate the parameter and then takes that evaluated value and calls the function. So if you pass here a+ 1 it will first make plus 1 as 4 and then pass that, if you do ++a it will increment a from 3 to 4 and pass 4. So, you are not going to get any of the pitfalls. But you get the advantage of the macro that is with the hash define macros you were able to avoid the overhead of function call all the parameter coping, then transfer of control, then the computation, and then again transfer of control back and the copy of the return value all this can be avoided because of compiler would try to actually put the x times x that I am doing here right at the site where the function has been called. So, that is the basic feature of the inlining.

(Refer Slide Time: 15:06)

The slide has a blue header with the title 'Macros & inline Functions: Compare and Contrast'. Below the title is a logo of a person in a graduation cap. The main content is a table comparing Macros and inline Functions.

Macros	inline Functions
<ul style="list-style-type: none">■ Expanded at the place of calls■ Efficient in execution■ Code bloats■ Has syntactic and semantic pitfalls■ Type checking for parameters is not done■ Helps to write max / swap for all types■ Errors are not checked during compilation■ Not available to debugger	<ul style="list-style-type: none">■ Expanded at the place of calls■ Efficient in execution■ Code bloats■ No pitfall■ Type checking for parameters is robust■ Needs template for the same purpose■ Errors are checked during compilation■ Available to debugger in DEBUG build

On the left side of the slide, there is a vertical sidebar with the following navigation links:

- Module 06
- Partha Pratim Das
- Objectives & Outline
- Context-aware & re-qualifier
- cover more
- Advantages
- Disadvantages
- inline function
- Macro
- inline
- Summary

At the bottom of the slide, there is a photo of a man (Partha Pratim Das) speaking, the text 'NPTEL MOOCs Programming in C++', and the number '20'.

So Inlining helps to get the benefit of macro to a good extent, while it protects the basic properties and the types of functions. If we can just compare them side by side then macros are expanded at the place of called inline are also in some way expanded the place of call. But it is not possible to show that to you because macro are expanded in text form so I could illustrate that to you, but inline functions are expanded in terms of the assembly code or the binary code at times, so it is not possible to understand it so easily, but we can we can just take it that it does the job right of the site were you have called. Both of them are efficient in execution both of them bloats the codes. The code becomes fatter, because what is happening you have one function defined and if you are doing a macro that function that macro may have been invoked at 10 places so the whole macro code will be copied ten times.

Similarly, inline function if I have function inline and that has been called at ten places the function body will occur at ten places so code will become bigger. Which may not be concern for some of the soft common software system that we do, but may be a concern in other cases like when we write program for a mobile phones and handled devices were the memory is really small if the program becomes larger then it becomes difficult to fit that app, because you are concern with the size of the app also. But we will not get much deeper into it.

Next point is the macros have syntactic and semantic pitfalls we have shown two of them, the `a + 1` was a syntactic pitfall so we could manage it by putting parenthesis around them, `++ a` was a syntactic pitfall because the way the plus plus is executed, so we could not manage that we have to leave it, but inline function will not have any such pitfalls.

Certainly macros do not check types for parameters, were as inline functions do that. So, their parameters are robust, so it is possible that a macro which intendedly written for say an integer type of value may inadvertently be called or invoked with some double variable, the compiler would not able to understand, whereas in inline function that is not possible because it is just the function that we have. However, my macros do have advantages for example, the fact that it does not check for type, help us to write some code were I do not know the type. For example, I want to write a code to swap two variables. Now, the code to swap two variables could swap int variables, could swap double variables, could swap char variables and so on.

But if you have to write in C++, the swap function, one swap function cannot do all this because if I have to swap two int variable then my parameters would be of int type, if I want to swap two double variables my parameters will be of double types, if I want to swap two pointers my parameters would be of pointer type. So I cannot write that function easily in C++, and inline that. Whereas, I could write one macro and swap it because macro is not checking that type, it will just check that there are two variables and third variables three assignments we know how to do swap it will just put that code here. So, macros do provide some advantages and going forwards towards the end of the course we will show that you can also do that with inline functions, but with the support of very later features of C++ known as templates; when you discuss that we will show how do this.

Finally, in for macros there is no error checking done for compilation in inline function the errors are checked and certainly like the manifest constant macros are also not visible to the debugger. You will not able to see the square macro if you try to debug because actually the compiler has never seen that square because it was replaced by the pre processor. Whereas, for inline functions you will be able to see all of that in the debug

build. If you do a debug build you will be able to see that.

The difference that I should highlight here, that with inline function 1 feature that the compiler do is if you do a debug build, the inlining is not done, if you do a release build then the inlineing is done. If you do a debug building an inline function is just like any other function, so you can debug into that function. Because in the debug build you are saying I want to debug I want to look inside what is happening whereas, in a release or production build you really want the efficiency to be there, you want that the code should run the fastest. So, you are not interested to debug any more you have already debug you know it is correct, that is when the inlineing actually happens, it is little bit subtle point, but please keep that this in mind over time you will slowly understand these factors. So, we would suggest that you always use inlining.

(Refer Slide Time: 20:29)

The slide has a blue header bar with the title 'Limitations of Function inlining'. On the left, there is a sidebar with a logo, the text 'Module 06 Partha Pratim Das', and a list of topics under 'Objectives & Outline'. A circular video player shows a person's face. The main content area contains a bulleted list of limitations:

- inlining is a directive – compiler may not inline functions with large body
- inline functions may not be recursive
- Function body is needed for inlining at the time of function call. Hence, implementation hiding is not possible. *Implement inline functions in header files*
- inline functions must not have two different definitions

At the bottom, there is footer text: 'NPTEL MOOCs Programming in C++', 'Partha Pratim Das', and '21'.

There are however some limitations that you should be aware off. Inlineing is called a Directive. A directive in a language is a suggestion to the compiler, you are telling the compiler look I think it is useful to inline this function, but it is not mandatory, it is not a binding on the compiler, it must inline. So you must say inline, but the compiler may not inline the function. If the compiler finds that inlineing it has problems or inlineing it does not really help the efficiency. For example as very simple example is, if a function body

is very large then what again you are doing by inlineing is that the function call is not required function written is not required, rest of it the computation has to be done any way.

If the function body is very large, then the additional overhead of call and return are very small, so you do not want to get into all this trouble of inlineing, but if the function body is very small inlineing really is. The compiler decides whether it wants to do inlineing or not. In many cases, in C++ the reverse is also true that you may not have said that a function to be inlined, but the compiler might inline it finding that it is efficient to inline.

Second point to be noted is inline functions cannot be recursive. What we are saying we are saying that at the place of inline, at the phase of function call we will put the body of the function if a function is recursive the body itself have another call. So, at that call you will again have to put the body of the function that will have another call. Now how many times you will put it that depends on how deep the recursion goes which you do not know till you have the values, till you know whether you are trying to do factorial 3 or you are trying to do factorial 77. You do not know how many times the inlineing has to happen, so recursion function cannot be inline they will necessarily have to be a normal function.

Since, inlineing is replacing the body. In the third point I highlight that fact that if you want to inline the function, then the function body will also have to be in header file. In the earlier in module 1 we have talked about source organization, where he said that all functions header must be prototype must be in headers files, dot h files function bodies implementation should be in separate dot CPP files, but for inline functions this is not possible because when an application sees the function prototype unless it can see the body how does it inline, how does it replace the body. So for inline function the body should also have to be in header. Certainly the inline functions must not have two different definitions because it does have two different definitions then the two invocations will have two different behaviors.

(Refer Slide Time: 23:48)

Module Summary

Module 06
Partha Pratim Das

Objectives & Outline

- const-ness & const-qualifier
- const-const Advantages - Pointers - Volatile
- inline Functions
- Macros (10 min)
- Summary

NPTEL MOOCs Programming in C++ Partha Pratim Das 22

- Revisit manifest constants from C
- Understand const-ness, its use and advantages over manifest constants
- Understand the interplay of const and pointer
- Understand the notion and use of volatile data
- Revisit macros with parameters from C
- Understand inline functions and their advantages over macros
- Limitations of inlineing

So, these are some of the limitations or restrictions of function inlineing that needs to be kept in mind. In summary, we have revisited the manifest constant for C; I am talking about the whole of module 6. So, we have revisited the manifest constants and we have understood the notion of const-ness as is available in C++ and we have seen, what the advantages of const over manifest constant are. We have also seen how const and pointer interplay. We have introduced the notion for volatility of data and seen how volatile data can be used in a C++ program.

Next, we have revisited macros with parameters form C and shown how inline functions or function inlineing can be used with advantage in place of macros which solves a number of syntactic and semantic problems that the macros offer. Finally, we have also looked at the restrictions on inline.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 10
Reference and Pointer

Welcome to module 7 of programming in C++. We will continue to discuss the procedural extension of C into C++. We have introduced that in module 6 and discussed the two basic concepts of cv-qualifier, constant and volatiles qualifier and the use of inline functions.

(Refer slide Time: 01:21)

The slide has a dark blue header bar with the title "Module Objectives". On the left, there is a vertical sidebar with a logo at the top, followed by the text "Module 07", "Partha Pratim Das", and "Objectives & Outlines". Below this, there is a list of topics: "Reference variable", "Call-by-reference", "Scope in C", "Scope in C++", "your Reference Parameter", "Return-by-reference", "References vs. Pointers", and "Summary". The main content area contains a bulleted list: "Understand References in C++" and "Compare and contrast References and Pointers". At the bottom of the slide, there is footer text: "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and the number "2".

In this module, we will introduce another concept called reference. This concept of reference is a closely related, but is very different from the concept of pointer. So, we will also; as we go through the module, we will also try to compare and contrast between the reference and the pointer. So, their objective is to understand reference and understand this contrast.

(Refer slide Time: 01:29)

The slide is titled 'Reference' and is part of 'Module 07'. It features a sidebar with navigation links like 'Partha Pratim Das', 'Objectives & Outlines', 'Reference variable', 'Call-by-reference', 'Scope in C', 'Scope in C++', 'const Reference Parameter', 'Return-by-reference', and 'Reference Pointers'. A video player shows a man speaking. The main content area has a blue header with the text 'A reference is an alias / synonym for an existing variable'. Below it, code is shown with annotations: 'int i = 15; // i is a variable' and 'int &j = i; // j is a reference to i'. Handwritten red arrows point from the text 'variable' to the variable 'i', from 'memory content' to the value '15', and from 'address' to the address '200'. Another arrow points from 'alias or reference' to the variable 'j'. The bottom of the slide shows a toolbar with various icons.

These are the different specific topics that we will go through. A reference is like an alias or a synonym for an existing variable. So, alias is like what we have in terms of our own names. We have some given name. We have some pet names and we can be referred, called it by either of the names. So, here reference variable also has that similar purpose. So, i is a variable which is declared. here it is initialized to 15 and in this context we have a variable i, which is defined. It has an initial value 15. Now, I define another variable and particularly look into this use of the & symbol here. We define a variable j and initialize it with i. Such a variable j is called a reference to i or a reference variable for i.

So, this reference variable is actually an alternate name and alias name for i. So, if I look into the memory, both i and j will actually represent the same memory location. So, if i happens to have a memory address 200, as I show here below if this is the address of i and its content is 15, the address of j will also be, 200 as of i. So, it is immaterial after this, this particular reference j has been introduced it is immaterial as to whether I refer i as i or I refer i as j. That is the basic concept of a reference.

(Refer slide Time: 03:40)

Program 07.01: Behavior of Reference

```
#include <iostream>
using namespace std;

int main() {
    int a = 10, b = a; // b is reference of a
    // a and b have the same memory
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    ++a; // Changing a appears as change in b
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    ++b; // Changing b also changes a
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    return 0;
}

a = 10, b = 10
a = 002BF944, b = 002BF944
a = 11, b = 11
a = 12, b = 12
```

So, let us concentrate on a small program to understand the behavior of reference. In this program, I will show you, there is a variable a and b is set as a reference of a. Then in the next two output statements, cout, we first print the value of a and b. we can see the output here; the 10 and 10; because a is 10 and b is a alias of a. So, if I print b, also I will print 10. And in the second line of the cout, we print that is the line here. We print the address of a, and we print the address of b. These are the addresses this line being printed. We can again check that they are identical addresses. That is they indeed are the same variable, just that they have two different names.

So, let us try to change the value of the variable. So, here we increment a, and then output it again. If we increment a, it will become 11, so you can see here that a, has become 11. And b, even though now operation was done with b, b also has become eleven. And, you can do it other way. If you increment b, then b becomes 12. And a, the variable to which b is a reference has also become 12. That is, they are very strongly coupled together and any one can be used for the purpose of the other.

(Refer slide Time: 05:36)

The slide has a blue header with the title 'Pitfalls in Reference'. On the left, there's a sidebar with navigation links: 'Module 07', 'Partha Pratim Das', 'Objectives & Outlines', 'Reference variable', 'Call-by-reference', 'Scope in C', 'Scope in C++', 'constant Reference Parameter', 'Return-by-reference', 'References', and 'Previous'. Below the sidebar is a portrait of Partha Pratim Das. The main content area contains a table:

Wrong declaration	Reason	Correct declaration
int& i;	no variable to refer to – must be initialized	int& i = j;
int& j = 5;	no address to refer to as 5 is a constant	const int& j = 5;
int& i = j + k;	only temporary address (result of j + k) to refer to	const int& i = j + k;

At the bottom of the slide, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

So, this is the basic motion. Now, certainly if you try to do this you will have to be careful that there are certain very typical pitfalls that you can get yourself into. So, three common pitfalls I illustrate here; that could be more. That is, if we just try to define a reference, but without initializing it with a variable. Then, the compiler will give you an error because a reference is an alias for some other variable. So, unless you define a variable along with it, initialize a variable along with it, there is no referent to refer to. Therefore, it is wrong.

So, if you just see on the table, on the left I show the erroneous declaration and on the right most I show the corresponding correct version, and you can understand the reason as given. If I look into the next one, that is, if I look in here, then you are trying to do a reference to a constant value. This is also is an error because a constant value is just a value. It does not have an address to reside.

So, you cannot have reference to a value, but you can have a constant reference to a value; because it is a constant value. So, the reference also will have to be a constant one. Otherwise, you can think of the danger that will happen; is this is related to the dangerous as we showed in conciseness. That if I, if this were correct, suppose this were correct, then j is 5.

Now, what will happen if I do `++ j`? Whatever it is referring to will get incremented. So, this will become; so, a constant 5 will become a constant 6, which is not possible. So, it has to be defined as `const`, so that you cannot make any changes to that.

Similarly, in the next one if you look in here I have expression `j+k`. and, I am trying to create a reference to that. But, the expression again does not have an address. The computation of `j + k` is stored only as a temporary location and those temporary locations are not retained. So, again I cannot create a reference to that. If I want to have a reference, that reference will have to be a constant, which will refer to the value of `j + k` as computed as the value of the reference.

So, you will not be able to change that because the expression `j+k` cannot be changed. So, if I make; if we allow a reference, then we would be able to change that; which is not semantically valid, if the reference is being made to an expression. So, all that you can say that. At this point of time, `j + k` had some value; which is treated as a constant. And, I have reference to that. So, all those reference will have to be constant. There could be more pitfalls, but these are the common one. So, I just choose to discuss them.

(Refer slide Time: 08:48)

The screenshot shows a presentation slide titled "C++ Program 07.02: Call-by-reference". The slide content includes:

```
#include <iostream>
using namespace std;

void Function_under_param_test// Function prototype
int ab, // Reference parameter
int c); // Value parameter

int main()
int a = 20;
cout << "a = " << a << endl << ab << endl << endl;
Function_under_param_test(a, a); // Function call
return 0;
}

void Function_under_param_test(int ab, int c) { // Function definition
cout << "b = " << b << endl << ab << endl << endl;
cout << "c = " << c << endl << bc << endl << endl;
}
```

Output:

```
a = 20, ab = 0023F430
b = 20, ab = 0023F430
c = 20, bc = 0023F96C
```

Annotations highlight the parameters `ab` and `c` in the function definitions and calls, and the output values `ab` and `bc`.

Now, why are we doing this? So, let me introduce a totally new concept of passing parameters to functions. We know how to pass parameters to function from C. It is called call by value. Just for a quick recap, the function as defined has formal parameters. When it is called, I have actual parameters. They correspond in the order of position and at the time of call, each actual parameter's value is copied to the formal parameter and then function is called. So, when the call is done, the function is in call. Actual parameters reside in some memory corresponding formal parameters reside in distinctly different memory. In contrast, we can do what is known as call by reference.

So, I would like you to focus on this particular line. We are trying, we have given a function header. Function under param test is a prototype. Look at this first parameter where we write the parameter name prefixed with &, which is the notation for reference. Such a parameter is called a reference parameter. I have also another parameter c in that function, which is the typical value parameter and we will follow the call by value rules.

Now, let us look into this part of use. So, use that and we choose to have one variable a, and call the function. That is, as actual parameter we pass a in place of both the formal parameters. Then, this is the definition of the function; where given the two parameters, we just print their value and we just print their addresses. Now, let us look into the output. So if we do this, then the first output will come from this slide before the function call, which is this output, which tells me a is at this location. And, this is the address. The second output will come from this cout, which prints b and gives the b's address.

Look at something very interesting. The address of b is exactly same as address of a, something that we do not expect in a call by value mechanism. And, to just show that what call by value would have done, you would look at the third output line cout and the corresponding output here of the parameter c. And, you do find that even though c also has the same value as of a, which it should, the address of c is different. So, this clearly show that between the two parameters b and c, c is following the original call by value rule by which is just the value of a which is copied to c. a and c continue to exist in two different memory locations, whereas, b has basically become a different name for a. It has become an alias for a.

These symptoms as we have just seen at the symptoms of alias that when I have two variables differently named, but they enjoy the same memory location and hence the same value. So, that is a reference variable. So when we use parameters, reference variables in parameters, we call them as call by reference mechanism. So, in call by reference the actual parameter and the formal parameter will have different names in the caller and the callee, but they will have the same memory location. So, this is the new feature that C++ allows us to do. Here, the all that I said are written at the bottom part of the slide. You can just read that.

(Refer slide Time: 13:15)

Call-by-value	Call-by-address
<pre>#include <stdio.h> void swap(int, int); // Call-by-value int main() { int a = 10, b = 15; printf("a= %d & b= %d to swap\n", a, b); swap(a, b); printf("a= %d & b= %d on swap\n", a, b); return 0; } void swap(int c, int d) { int t; t = c; c = d; d = t; } </pre> <ul style="list-style-type: none"> • a= 10 & b= 15 to swap • a= 10 & b= 15 on swap • Passing values of a=10 & b=15 • In caller; c = 10 & d = 15 • Swapping the values of c & d 	<pre>#include <stdio.h> void swap(int *, int *); // Call-by-address int main() { int a=10, b=15; printf("a= %d & b= %d to swap\n", a, b); swap(&a, &b); printf("a= %d & b= %d on swap\n", a, b); return 0; } void swap(int *x, int *y){ int t; t = *x; *x = *y; *y = t; } </pre> <ul style="list-style-type: none"> • a= 10 & b= 15 to swap • a= 15 & b= 10 on swap • Passing Address of a & b • In callee x = Addr(a) & y = Addr(b) • Values at the addresses is swapped

Now, we will just preside for a while. We have just learnt, what is call by reference mechanism. But, we will still wonder as to why we are trying to do this. So, to appreciate why we are trying to do this, I take an example from C. And, this is one example which every one of you, who have done little bit of C would know I am trying to write a swap function. It will, which will take two variables a, and b and try to swap them. So, this is what we are focusing on. This is a signature.

If I write this function, if I write this code I am calling swap here and this print out at to show what is the value of a and b. So, this is a first print which comes from this particular printf; which shows that a, and b to swap have values 10 and 15, as they are

initialized. Then, I go to swap. So, c and d becomes 10 and 15, the code of swap swaps them and back, I print again. But, unfortunately the values, actual parameters have not been swapped. So, the swap did not work and this is to be expected. Because what is the mechanism? The mechanism is call by value; which means that when I have called the function swap, the function has taken the value of a copied it to c, taken value of b copied it to d. And then, the function has done whatever it had to do. It had swapped c and d. But, these c and d have different locations than a, and b. So, nothing has happened to a, and b. They are well protected as actual parameters. So, when the function comes back a, and b are as same. They have not been swapped. So, swapping does not work this way.

So, we have learnt in C we cannot write the swap this way. So, what do I have to do? I have to do some tricking around; some tricks. So, what trick do we do? The trick that we try to do is define swap in this way. I have mentioned this as called by address; call by address is not a very formally accepted name. It is actually called by value, but the only difference is that here the parameter that we pass is a pointed type parameter.

So, we say instead of swapping two integers, now we will swap two pointers to integers. Therefore, this code is also written with the pointers where, and since these are the pointers. Whenever I have to refer to the first, I have to it $*x$ now and the second by $*y$ and we will do that. So, since these are pointers, if I have to call them at this slide, the two actual parameters a, and b will have to be passed as the address of a, and address of b. So, we pass the two addresses.

Now, what is happening, If I look at x ; is a pointer to a. y is a pointer to b. Now, what it is doing? It is trying to swap the value of $*x$ and $*y$. It is not changing the value of x and y. It is not swapping these. It is swapping the values of $*x$ and $*y$. What is $*x$? $*x$ is actually a. If x is the address of a, then $*x$ is actually a. So, when I swap $*x$ and similarly $*y$, it is actually b. So, when I swap $*x$ with $*y$, I am actually swapping a with b.

So, what I am doing is, basically since call by value will not allow me to make changes to the actual parameter, I am sending their addresses out and remotely I am allowing the swap function to use the address to actually refer to the actual parameters, and then make

the change. So, in a way I am breaking the rule of call by value because without that I cannot bring the changes back.

So, here we will have; we see that. Since we are not being able to swap the values directly, we use the call by address kind of mechanism, where we pass the addresses and access those values through those addresses. Certainly, it is kinds of a back door to get, achieve the result. And, why are we doing this? What is the fundamental symptom for which we have to do this? Swap may be one specific instance. The symptom is if we do call by value, then the actual parameter is copied to the formal parameter. So, whatever you do in the function with the formal parameter, those effects will never come back to the actual parameter. This is what we wanted. But, in swap we need the changes in the formal parameter to come back to the actual parameter. Only then, swap can happen. If I am swapping a, and b, unless a can change and unless b can change, swap will not happen.

So to generalize, call by value allows us to have only input kind of parameters; whether values can go from the caller function to the called function. But, call by value does not allow me to do output kind of parameter; that is, I cannot compute a value in the function and get it back to the parameter. So, as such function returns only one value. So, if I want a function where from which I need more than one output, I have no mechanism, straight mechanism in C. Call by value fails. So, the only other mechanism is to use the addresses and do it in a round about way.

Now, let us see. Now, let us try to combine the two factors we have talked up. One, we have introduced the concept of reference, call by reference and we have talked about the difficulty of having multiple output from a function, from a C function. So, let us look at particularly in the swap example again. Now, let us. On the left hand side, we have the call by value example. The example that we saw is actually wrong because it cannot change because it should not change. This is the call by value prototype.

Now, here in C++ all the only change that we have made is instead of having call by value. We are now saying that we have two parameters, which are called by reference. That is the parameters are not usual parameters. They are reference parameters. And,

now you write the whole thing with the reference parameter. Rest of it, rest of the code between what you see in the C and what you see in C++ are same. These are the just two places where changes are made.

What is the effect? If it is a call by reference, then when this particular call happens, the actual parameter a, and the formal parameter x here, they are, they enjoy the same location; x is another name for a, y is another name for b. So, if I interchange x and y it is same as interchanging a, and b. So, I can get the whole effect in my code. So, what I gain by doing this? Several things; one is I do not need to take the back door. If I want a parameter to be input, I make a call by value; if I want the parameter to be output as well, then I will do a call by reference because then the changes done in that formal parameter within the function will also be available in the actual parameters; because the actual parameter and the corresponding formal parameter enjoys the same address.

So, that is the advantage of using reference and doing the call by reference mechanism in C++. It certainly makes the code much cleaner. It makes it more easy and reliable to write programs. And, as we will see later on, it also often saves a lot of effort because in call by value you have to make a copy of that value. Now, as long as your value is an integer, maybe making a copy is not costly. But, think of, if you are, if the parameter that you are passing is possibly a structure which may have say 10 kilo byte of size having 200 different members, 200 different components, then copying that itself may be a lot of cost. So, you would not like to incur that cost. You could just use reference.

There is a side effect of using reference parameter. Now, said that, if we do a reference parameter, that is if we use call by reference, then the parameter actually is input and output parameter. It is input because, when the moment I am calling whatever the value that the actual parameter has, the formal parameter will also have the same value. So, it serves the purpose of input. So, actually I do not need to do call by value. I can only use call by reference. But, if I do call by reference, then the potential is, that any changes made to the formal parameter will also get reflected in the actual parameter. So, any inadvertent change done in the formal parameter or intentional change is done in the formal parameter, will spoil my actual parameter. So, can we do something so that I can

use call by reference, but just make it an input parameter. So, to do that, you make use of the const.

So, what here? Just look at the code here. Particularly focus on this function, which is taking one formal parameter `x` of type `int`. But, what we have done? Before that we have said it is `const`. So, what does that mean? Because it is a call by reference because it is a reference parameter, so when I call it at this point as `a`; `a` and `x` refer to the same address. But, I am saying that the reference `x` is constant; which means that no change of `x` is possible. This is a situation which is very similar to what we discussed in case of constness of pointer and the conciseness of pointed data. We are sure that the data itself may not be constant, but if I am holding a constant pointed to that data, then certainly that pointer will not allow me to change the data. So, similarly here I have a constant reference or rather I have a reference to the conciseness of the data. So, `x` will not allow me to change.

So, if within the function, I try to write something like `++ x`. This code will not compile because `++ x` being that I am trying to change the value of `x`. But, `x` is a constant reference. `x` is a constant value. So, whatever it got initialized with. What did it get initialized with? It got initialized with `a`. When the call was made that cannot be changed. So, now what happens? At the time of call the value of `a` will be available as a value of `x`. That is the same address. But `x`, you cannot change within the function. So, no way changes in `a` `x` can affect the actual parameter `a`. So result, the effect cannot come back. So, now, the parameter becomes purely an input parameter.

So, using a reference we can make either input output parameter or we can make input only parameter, if we make that a constant reference parameter. So, instead of using call by value, in a large number of cases we will try to use just constant reference parameter; because you would not need to do copy. And, we will still be protected that our actual parameter will not get affected.

So, on the right hand side we just show that, what is a proper way of doing it, where `x` is a constant reference and you do not try to. Here, we were trying to increment `x` and then return that. Here, we do not do that. We compute `x` with adding one with it and then

return that so. The code on the right hand side will compile and run well. There is no violation. The code on the left hand side will have compilation error and we will not be able to proceed with that any further. At this point, we will stop and we will continue in the next part.

Programming in C++
Prof. Partha Prathim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 11
Reference and Pointer (Contd.)

Welcome to module 7 in Programming in C++. We have discussed the basic concept of reference and using that we have introduced the notion of call by reference. And, we have shown that how certain functions like swapping can be written in a better way in C++ by using the call by reference mechanism. We have also discussed that a reference parameter can be a general input/output parameter for a function. Therefore, if I do not want the function to change the formal parameter and make consequent changes to the actual parameter, then we also have the option of defining a reference parameter to be a const parameter by which it can only be initialized by the actual parameter. But, any changes made to that will not be allowed by the compiler. We will now next continue and talk about the other side of the function mechanisms.

(Refer slide Time: 01:29)

Return-by-value	Return-by-reference
<pre>#include <iostream> using namespace std; int Function_Return_By_Val(int &x) { cout << "x=" << x << endl; return (x); } int main() { int a = 10; cout << "a=" << a << endl; const int& b = // const needed. Why? Function_Return_By_Val(b); cout << "b=" << b << endl; return 0; } a = 10 x= 00DCFD1B x = 10 x= 00DCFD1B b = 10 b= 00DCFD1B</pre>	<pre>#include <iostream> using namespace std; int& Function_Return_By_Ref(int &x) { cout << "x=" << x << endl; return (x); } int main() { int a = 10; cout << "a=" << a << endl; const int& b = // const optional Function_Return_By_Ref(a); cout << "b=" << b << endl; return 0; } a = 10 x= 00A7FBFC x = 10 x= 00A7FBFC b = 10 b= 00A7FBFC</pre>

In C, we use the mechanism to get a value back from a function; is called return by value mechanism. That is, as parameters are copied from actual to formal, the return value is also copied from the return value expression, which we write in the return statement back to wherever we are assigning that function value. So, in contrast in C++ it is possible to

return a value by reference. Let us see what we are saying.

So, first let us focus on the two sides. On left, we have return by value; on right, we have return by reference. So, on left we have a function which is a typical C return form in C++. It is only using a call by reference. And on right, please note that after the return type we have the reference symbol. So, if it says that we are returning the reference of the return expression. And then, the use is similar. Here, we invoke this function; here we again invoke this function. And to keep the result of the function, and this is primarily for the purpose of illustrating the effect to you.

I have used another reference variable b. So, b will keep the reference to the value that is returned. If you look into the return by value part that is on this side, you will need to understand that this reference b must be a constant. Why should this be a constant? because if you recall the pitfalls of reference definition, we had shown that if I have an expression j+k, I cannot create a reference to that because j+k is computed in a temporary.

Now, here what do I have on the right-hand side? I have a function invocation. What is a function invocation? It is an expression. So, I cannot write a difference to that. I can have to write a constant reference to it, which is the temporary location which will be preserved through this reference. On this, writing the constant is just for the sake of uniformity. It is not actually required. You will understand that once you understand mechanism.

Let us look at the output now. So, the first output that will come the cout which is here, which prints a, and its address. Next is within the function. The function has been called. After this print, the function has been called here. So, next is this cout, which is this output where you print x. It is a call by reference. So as expected a, and x have the same address.

Finally, you look at this third output which will come from main, after the function has returned. And, what does a function do? Function simply returns a value that you had passed it. So, it is expected to return the same value. So, b returns the same value. But, if you print address of b here, it is different from the address of a, or address of x. And, this is expected; because it is returning by value. So, what it is returning is a copy of x; which is in a temporary, and I have held that temporary is a part of b; as a reference in b.

(Refer slide Time: 05:52)

Module 07
Partha Pratim Das
Objectives & Outcomes
Reference variable
Call by reference
Topic in C
Topic in C++
Topic Definition Parameter
Return-by-reference
Return-by-Value

Program 07.06: Return-by-reference

Page 39 / 39

Return-by-value	Return-by-reference
<pre>#include <iostream> using namespace std; int Function_Return_By_Val(int x) { cout << "x=" << x << endl; return (x); } int main() { int a = 10; cout << "a=" << a << endl; const int& b = const optional Function_Return_By_Val(a); cout << "b=" << b << endl; return 0; }</pre>	<pre>#include <iostream> using namespace std; int& Function_Return_By_Ref(int& x) { cout << "x=" << x << endl; return (x); } int main() { int a = 10; cout << "a=" << a << endl; const int& b = const optional Function_Return_By_Ref(a); cout << "b=" << b << endl; return 0; }</pre>
<pre>a = 10 a= 00DCFD18 a = 10 x= 00DCFD18 b = 10 b= 00DCFD18</pre>	<pre>a = 10 a= 00A7FBFC a = 10 x= 00A7FBFC b = 10 b= 00A7FBFC</pre>

Think about the similar on the reference side. Here, we are looking at call by, return by reference. The first output comes from here which is, a and its address. Second is from the function which is x and its address. Call by reference; they have to be identical. they are indeed identical. The third output comes from here after the function has returned. You see that it is not only b is same, this address of b is also same; because what has actually been returned, what has been returned is not the value of x.

But, what has been returned is a reference that is the address of x itself; the reference of x, which is an alias of x. So, b now becomes an alias of what was x. And, what was x? x was an alias of a. So, b has in this process become an alias of a. So, that is the difference between return by value and return by reference. Again, if you do return by reference, then it will show some interesting and tricky issues. But if you do return by reference, then again, we can avoid copying of large structures if that is what we want to return. So, in many places you may want to return by reference.

(Refer slide Time: 07:22)

Module 07

Partha Pratim Das

Objectives & Outcomes

Reference variable

Call-by-reference

Scope in C

Scope in C++

over Reference Parameter

Return-by-reference

References vs Pointers

Summary

Page 41/41

Return-by-reference

```
#include <iostream>
using namespace std;
int Return_Ref(int &x) {
    return (x);
}

int main() {
    int a = 10, b;
    b = Return_Ref(a);
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    Return_Ref(a) = 3; // Changes
    cout << "a = " << a;
    cout << endl;
    return 0;
}

a = 10 and b = 10
a = 3
```

Return-by-reference – Risky!

```
#include <iostream>
using namespace std;
int Return_Ref(int &x) {
    int t = x;
    t++;
    return (t);
}

int main() {
    int a = 10, b;
    b = Return_Ref(a);
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    Return_Ref(a) = 3;
    cout << "a = " << a;
    cout << endl;
    return 0;
}

a = 10 and b = 11
a = 10
```

Note how a value is assigned to function call
We expect a to be 3, but it has not changed

Now as I said, return by reference can get tricky at times. Look at here. There is a function which takes a parameter by reference and returns that same parameter by reference. And, look at this line and first we will have to believe your eyes that what you are seeing is correct program. You have never seen a function invocation occurring on the left-hand side of an assignment. Function invocation is always on the right-hand side. So, here this is valid; because what is the function returning? The function is actually returning an alias. It is actually returning an address; it is actually returning a variable. So, if I say that I am making an assignment 3 to it, then I am actually making a assignment to that alias.

So, let us see what it means. Let us look at the output. We have the outputs here. So, this is the cout, the function was called at this point with a, x is 10, which is, a is 10. So, it returned the same x. So, b has become 10. So, when I look at this output, it is a is 10, b is 10. No surprise. I call it once more here. And, assign 3 to the return reference. So, what will it mean? if I am calling it with a, then a and x become alias. x is an alias of a. I am returning by reference. So, what I returned is an alias of x, which has to be an alias of a. So, what I have got here as return is actually an alias of a. So, you can always make an assignment 3 to it. Where will that assignment happen? It is an alias of a. So, assignment will happen in a. So, check a out of this. a has become 3. So, this is what gets possible if you do a return by reference.

Now, we will subsequently as we go forward, to get to later features, we will show how this kind of a tricky thing and little bit confusing thing can be used for advantage in some places in the program. I am not advising that you write this kind of code very frequently. But, there are places where this can be used to advantage in a good way.

(Refer slide Time: 10:44)

Return-by-reference	Return-by-reference – Risky!
<pre>#include <iostream> using namespace std; int Return_ref(int x) { return (x); } int main() { int a = 10, b; b = Return_ref(a); cout << "a = " << a << endl and b = " << b << endl; Return_ref(a) = 3; // Changes // reference cout << "a = " << a; return 0; }</pre>	<pre>#include <iostream> using namespace std; int Return_ref(int x) { int t = x; t++; return (t); } int main() { int a = 10, b; b = Return_ref(a); cout << "a = " << a << endl and b = " << b << endl; Return_ref(a) = 3; // Changes // reference cout << "a = " << a; return 0; }</pre>
<p>a = 10 and b = 10 a = 3</p>	<p>a = 10 and b = 11 a = 10</p>
<ul style="list-style-type: none"> • Note how a value is assigned to function call • We expect a to be 3, but it has not changed 	

And, using this on the right column which you had not seen so far, I show that if you are not skilled, you might shoot at your foot in trying to do this. Look at this code, which is very similar to what the earlier code was. The only difference being that it now takes the parameter x, which is naturally an alias of a, because it is call by reference and it has a local variable t and it initializes t with x and then increments t.

And then, look at here, earlier it was returning x. Now, it returns t which is this local variable as a reference. And, you did a same thing here. The first two lines are at similar. They produce a same output. So, there is nothing to look at. Look at this line. You have done this here. So, if you do this and try to expect that, some change will happen to a, or something like that. You will be surprised that nothing will happen; because what you have done? You have actually returned a local variable. So, what gets returned here is a reference to the local variable t. And that is very risky; because once you have returned the reference, the function call is over. So, the local variable has disappeared, but your reference is still alive. So, your reference says that I have the variable. The function call has terminated. So, it actually does not exist. This variable t no more exists, that is dead.

But, you hold a reference to that. So, the results of this kind of program could be very unpredictable.

So, the bottom line prescription is if you are using return by reference, never return a local variable. If you are returning by reference, always return a variable which is available, which should logically be live after the function call has ended. So, those variables could be global, those variables as we will see could be static members, those could be the actual parameters that you got as alias in the formal parameter, those could be dynamically allocated values and so on. But, not the automatic local variables that a function has, because that can get you into real difficulty and these are bugs which are very difficult to detect; because you will not see anything in the code. On the code, it looks everything clean and simple. But, still your results will become unpredictable.

(Refer slide Time: 13:32)

The screenshot shows a presentation slide with a blue header containing the title 'Difference between Reference and Pointer'. The slide is part of 'Module 07' by Partha Pratim Das. The content is organized into two columns: 'Pointers' and 'References'. The 'Pointers' column lists:

- Refers to an address
- Pointers can point to NULL.
int *p = NULL; // p is not pointing
- Pointers can point to different variables at different times

```
int a, b, *p;
p = &a; // p points to a
...
p = &b // p points to b
```
- NULL checking is required
- Allows users to operate on the address - diff pointers, increment, etc.

The 'References' column lists:

- Refers to an address
- References cannot be NULL.
int &j; // wrong
- For a reference, its referent is fixed

```
int a, c, &b = a; // Okay
.....
&b = c // Error
```
- Makes code faster
Does not require NULL checking
- Does not allow users to operate on the address. All operations are interpreted for the referent

Pointers	References
• Refers to an address	• Refers to an address
• Pointers can point to NULL. int *p = NULL; // p is not pointing	• References cannot be NULL. int &j; // wrong
• Pointers can point to different variables at different times <pre>int a, b, *p; p = &a; // p points to a ... p = &b // p points to b</pre>	• For a reference, its referent is fixed <pre>int a, c, &b = a; // Okay &b = c // Error</pre>
• NULL checking is required	• Makes code faster Does not require NULL checking
• Allows users to operate on the address - diff pointers, increment, etc.	• Does not allow users to operate on the address. All operations are interpreted for the referent

Finally, as we have seen through this module, reference is talking about an alias of a variable it is a mechanism which allows you to change the value of a variable without actually taking the name of the variable. So, in that way it has got lot of similarity and differences with the pointers.

So, I will just summarize this difference in these two columns. So, between the pointers and references, both refer to addresses. Pointers refer to address; reference also refers to an address. So, they are similar to that extent, but they differ in multiple ways. For example, look at the next point. Pointers can point to NULL; I can assign a pointer to

null. What it means? Semantically, it means that I am not pointing anywhere. It has no pointed data that I carry. But, reference cannot be NULL; because it is an alias. It is just an alias. So, it has to have a variable to effect. So, that is a basic difference.

Since pointer points to different other data, unless a pointer is constant, I can actually change the data that it is pointing to. So, the example clearly shows that you can do that. If p was initially pointing to a, then at some point statements, it can point to b. But, a reference; for a reference, what you are referring to is fixed by the definition; because it is an alternate name for a variable. So, certainly I cannot change that reference and make the name different.

For example, if we just look into, into this line, if you are trying to do like this, then you will not be able to achieve that if you write it as &b assigned c, thinking that you will change the reference of b, from a to c. You will not be able to do that because a moment you write &b. Since, it is an alias for a, &b will be understood as &a, because this is an alias. So, whatever you write to as b, what should apply to a, because this is an alias of a. And then, &a is an address of a; & is the address operator.

So, what you saying that; you trying to assign c to the address of a, which is the meaningless thing. So, you can try in all possibilities. For example, if you try to say okay, am, I would like to change the reference by putting c to b, but then b is a. So, if you try to assign c to b, you are basically assigning c to a. There is no way the language does not give you any mechanism, any operator to do anything with the reference. You can only refer. And, whatever you do actually is interpreted in terms of the data that is referred, the referent. So, that is a basic difference.

For pointers, since it is possible that I am not pointing anywhere. Before using a pointer, we need to check for null. Reference does not need that because it is, if it exists, then it is alias of something or it does not exist. So, it makes the code much easier to write. You do not have to bother about that and certainly it makes it faster in that sense because those checks are not required.

Finally, if you look into exactly if both pointers and references are addresses, and exactly what is the difference? The difference is not in the empowerment, not in terms of what you get for reference. The difference is in terms of what you do not get for reference. If you have a pointer variable, then you are given with a number of operators that you can

operate with that variable. You can actually use that address as a value and do different things. You can add an integer to it and advance a pointer. You can take two pointer values and make a difference and see how many elements exist between these two pointers in an array.

So, all these operators are given to you, by which you can change the pointer in whatever way you want; which makes pointer so powerful. In reference, also the address is stored. But, there is no way that you can catch hold of that address. You cannot hold that address. There is no operation that is given on the reference. Any operation that you try to do, actually boils down to operating on the referred object or the referent. So, that is the basic difference.

And, in the way C++ is designed, both pointers and references have their own respective place. It is not possible to do away with pointers and reference really add a value, but there are languages where you do not have both these mechanisms. So, it is good to understand. Particularly, if some you know java you would know that Java has a reference, does not have a pointer. And, I am just raising this point because I want you to note that the reference in java is not like the reference in C++. So, if you thought that let me have more clarification in the concept of reference and read the favorite java book, then you will only get more utterly confused. So, there is no real parallel of the reference of java in C++, but it is mostly similar to constant pointer. What you have in java is a reference is actually a pointer, but it is a constant pointer that you cannot change it. But, it is neither fully a pointer nor fully a reference in C++. And, there are languages; like c does not have reference.

There are languages which do not have pointers and so on. In C++ we have both. So, we have a big responsibility to decide in a given context, whether we need to use the pointer or we need to use a reference. And as we go along, different features and different programming examples will keep on highlighting that the choice has to be judicious and right for you to become an efficient C++ programmer.

So, in this module 7, we have introduced the concept of reference in C++ and we have studied the difference between call by value and call by reference. We have also introduced the concept of return by reference and studied the difference between the two mechanisms. We have shown some interesting pitfalls, tricks, tricky situations that may

arise out of this and we have discussed about the differences between references and pointers.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 12
Default Parameters and Function Overloading

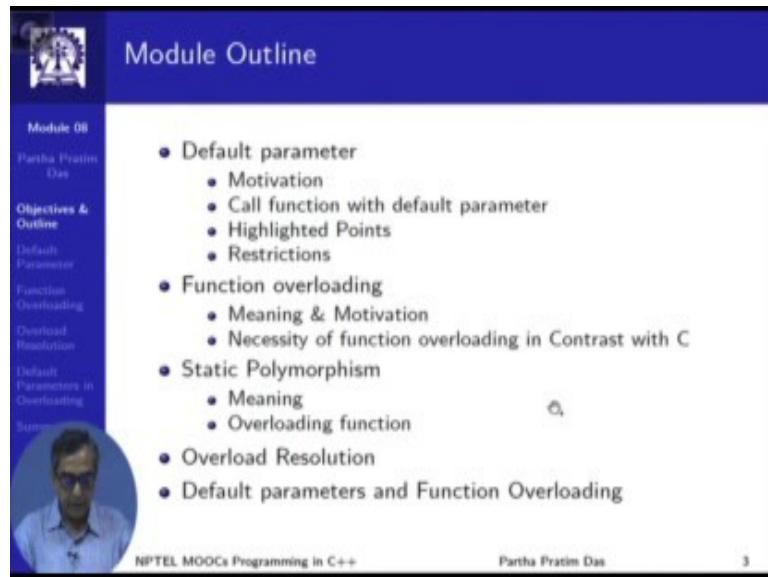
Welcome to module 8 of Programming in C++. We have been doing the better C features of C++, we have already talked about Const and Volatile and Macros, Inline functions and also the reference variable and call by reference and return by reference mechanisms. These are all better C features of C++. We will now in this module 8, talk about Default Parameters and Function Overloading.

(Refer slide Time: 01:03)

The slide has a dark blue header bar with the title "Module Objectives". On the left, there is a vertical sidebar with a logo at the top, followed by a list of navigation links: "Module 08", "Partha Pratim Das", "Objectives & Outline", "Default Parameter", "Function Overloading", "Overload Resolution", "Default Parameters in Overloading", and "Summary". The main content area contains two bullet points under the heading "Module Objectives": "Understand default parameters" and "Understand function overloading and Resolution". At the bottom of the slide, there is footer text: "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and the number "2".

The are two very core very important features which go a long way in enabling the object oriented properties of C++ language.

(Refer slide Time: 01:19)



The slide shows a module outline for 'Module 08' by Partha Pratim Das. The outline includes sections on Default parameter, Function overloading, Static Polymorphism, Overload Resolution, and Default parameters and Function Overloading. A video thumbnail of the speaker is visible on the left.

Module Outline

Module 08
Partha Pratim Das

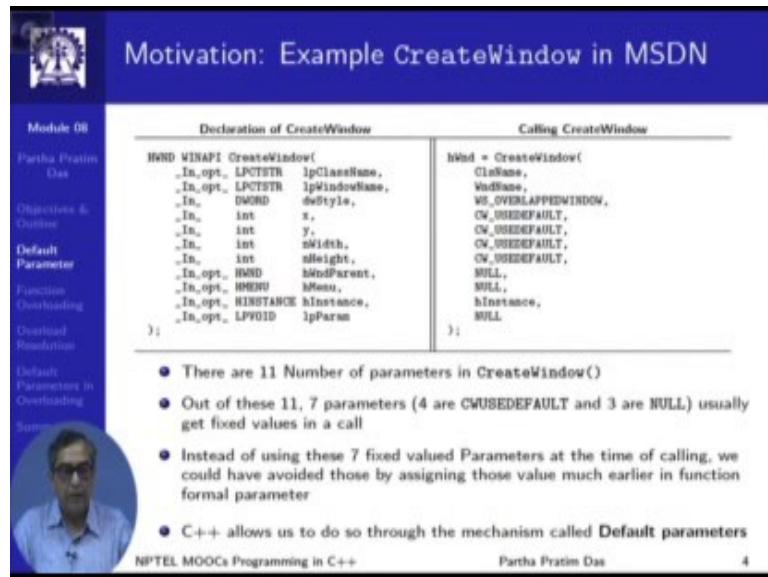
Objectives & Outline
Default Parameter
Function Overloading
Overload Resolution
Default Parameters in Overloading Summary

NPTEL MOOCs Programming in C++ Partha Pratim Das 3

- Default parameter
 - Motivation
 - Call function with default parameter
 - Highlighted Points
 - Restrictions
- Function overloading
 - Meaning & Motivation
 - Necessity of function overloading in Contrast with C
- Static Polymorphism
 - Meaning
 - Overloading function
- Overload Resolution
- Default parameters and Function Overloading

This will be the overall outline for this module which you continue to see on the left panel of the presentation. So, we now get started.

(Refer slide Time: 01:32)



The slide shows the 'Motivation: Example CreateWindow in MSDN' section. It displays the declaration and calling code for the Windows API 'CreateWindow'. Below the code, a list of bullet points explains the use of default parameters in this context.

Motivation: Example CreateWindow in MSDN

Module 08
Partha Pratim Das

Objectives & Outline
Default Parameter
Function Overloading
Overload Resolution
Default Parameters in Overloading Summary

NPTEL MOOCs Programming in C++ Partha Pratim Das 4

Declaration of CreateWindow	Calling CreateWindow
<pre>HWND WINAPI CreateWindow(_In_opt_ LPCTSTR lpClassName, _In_opt_ LPCTSTR lpWindowName, _In_ DWORD dwStyle, _In_ int x, _In_ int y, _In_ int nWidth, _In_ int nHeight, _In_opt_ HWND hWndParent, _In_opt_ HMENU hMenu, _In_opt_ HINSTANCE hInstance, _In_opt_ LPVOID lpParam);</pre>	<pre>hWnd = CreateWindow(ClassName, WindowName, WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, NULL, NULL, hInstance, NULL);</pre>

- There are 11 Number of parameters in CreateWindow()
- Out of these 11, 7 parameters (4 are CW_USEDEFAULT and 3 are NULL) usually get fixed values in a call
- Instead of using these 7 fixed valued Parameters at the time of calling, we could have avoided those by assigning those value much earlier in function formal parameter
- C++ allows us to do so through the mechanism called Default parameters

Now, first we will discuss about default parameters and I start with an example from C particularly this is an example from MSDN, the windows programming. If you are

familiar with a graphics programming and windows you would have seen this function, if you have not seen it, it really does not matter. All that I want to show on the left column is the prototype or header of the create window function. You can see the function has a large number of; specifically, 11 different parameters for its specification in the prototype.

When you need to create a graphics window you will need to specify all these 11 parameters to call the create window function. On the right column I show a typical create window call for creating window. Now, if will consider this particular function you can see two things; one is it has a large number of parameters, 11 parameters and to be able to use this you will typically need to know what these 11 parameters mean and you will need to specify those parameters.

Now, often you just want to create a sample window, you just want a window which has a default background color, default text color, you want it at a default center location in the monitor, you wanted it to have a default width and height and so on. Given that you may not be necessary that you will specify all distinct values; distinct for your applications in these 11 parameters. So, to facilitate that if you look into the actual call you will see that in the call there are several parameters like; if you look at the parameters here, these are parameters which are basically manifest constants, CW stands for create window use default. It is saying that there is some default value already defined in the library so you can use those values. So basically, you are not supplying those values.

Similarly, if you look into the window parent and the menu or if you looking into the window parameters we are just passing null which are basically again a kind of a default value. hInstance, the instance of the window. if you know window programming you would know is an instance of your application which also kind of gets a default values so is the same with the type of window that you are creating that is you are creating a overlapped window and so on.

Even though these 11 parameters are there in the function and you need to specify all of them several of them are given default values by the library or are passed as null and you

actually need to specify few things like what is your window class which defines what a specific structure you are doing and possibly the name of the window to make this call. But yet, you need the full range of all eleven parameters to be actually written in the call.

It would have been good that if you could arrange the parameters in a way so that only the parameters that you need to specify for your call need to be given at the call sight. If the system could understand and take the usually default parameters without your having to specify some default values for them, every time you make that call. So, C++ allows us to do something regarding this and that feature is known as Default Parameter.

(Refer slide Time: 05:50)

```
#include <iostream>
using namespace std;

int IdentityFunction(int a = 10) { // Default value for the parameter
    return (a);
}

int main() {
    int x = 5, y;

    y = IdentityFunction(x); // Usual function call
    cout << "y = " << y << endl; // y = 5

    y = IdentityFunction(); // Uses default parameter
    cout << "y = " << y << endl; // y = 10
}
```

Now, here is an example to just illustrate what default parameters mean. Consider this function, the functionality is not really very important, we are calling this as IdentityFunction, that, it takes a parameter and simply returns the same parameter back. But what we are interested to highlight, what you will need to observe here, is a parameter has a value given in the signature of the function as kind of an initial value of the parameter a. Now, we understand what initial value of a variable within a certain scope are, what is a meaning of having a kind of initial or default for a parameter. To understand that let us, look into the call, let us look into to the first call of the function.

First call of the function is usual we are calling the identity function with actual parameter x which currently has a value 5, so we are passing that parameter here and therefore it will return the same value and as you output from this line and you will get first line of the output at this point. There is no surprise till this point.

Now, let us focus on the second call of the same function. You will be surprised to note that now I can call this function without passing any parameter. We have so far in C, always known that the number of actual parameters and the number of formal parameters must match between the call and the prototype of the function. The actual parameters and the formal parameters must match in their order, in their data type and so on. But here, the function actually has one parameter, but when I call here I am not specifying that parameter. This is what the default parameter feature is. What the C++ does? since you have made the call without any parameter and since the function is already defined with a parameter value default value 10 here it will assume that you have as if called the function with this default value.

So, if you default a parameter then you actually get two options; one, as you get here where you can make the function call exactly in the way you have been making function calls earlier, specifying if actual parameter, or you can make a choice using the default value of the parameter and not need to specify an actual parameter you can just skip that. the system will understand that you are using the parameter and specify the value which is given in the default in the function declaration itself. So, this is the basic feature of default parameter. Let us now proceed and take a look into another example.

(Refer slide Time: 09:13)

Module 08
Partha Pratim Das
Objectives & Outline
Default Parameter
Function Overriding
Overload Resolution
Default Parameters in Overloading
Summary

```
#include<iostream>
using namespace std;

int Add(int a = 10, int b = 20) {
    return (a + b);
}

int main(){
    int x = 5, y = 6, z;

    z = Add(x, y); // Usual function call -- a = x = 5 & b = y = 6
    cout << "Sum = " << z << endl;

    z = Add(x); // One parameter defaulted -- a = x = 5 & b = 20
    cout << "Sum = " << z << endl;

    z = Add(); // Both parameter defaulted -- a = 10 & b = 20
    cout << "Sum = " << z << endl;
}

-----
Sum = 11
Sum = 25
Sum = 30
```

In the last example, we showed one function which has a default parameter and now we are showing that it is not necessary that you will have only one default parameter you can actually have up to default parameters or any arbitrary number of default parameters also. We just show a function here, please do not become concerned about the functionality of this, the algorithm for this, it simply takes two parameters adds them and returns them, just look at the definition of this function here, and it has two parameters both int; int a and int b. And what we have done is, each of the parameters have been defaulted with an initial value. So, with that, when we use this function and make a call if you look at the first call here, the first call is a usual function call, x and y are two actual parameters. We call add(x,y), so x corresponds to 'a' gets copied there, it is call by value as you can see, y is copied to b and function is called with the values of x and y that is 5 and 6 and therefore when you print the sum here we get the first output that is the sum turns out to be 11.

Now, look at the second call of the same function. In the second call of this function, we provide the first parameter x but we have not provided the second parameter, which means that though the function actually had two parameters we are calling it with one parameter. So, what will happen? This one parameter that we have provided that will correspond to x will correspond to the formal parameter a, so whatever is the value of x

will get copied in the call by value to the formal parameter a. But the second parameter has not been provided, so the system will understand that you are using the default value of the second parameter, that is, the second parameter will be taken to be 20 because a default value given in the second parameter is 20. Therefore, in this case the function will proceed with ‘a’ being 5 and ‘b’ being 10 and it will compute and you can see from the output, a+ b will compute to be 25 and that is what will be printed.

We can extend this further. This is what we do in the third instance of the call where we have not provided any actual parameter. So, the two parameters of the function a, and b both are taken their default values, that is ‘a’ is 10 and b is 20 and the function here it is not really visible, but the function here prints 30 as some result. We can see that, it is not necessary that I should default only one parameter, but it is possible that I can default an arbitrary number of parameters in a function. And this is what will be very useful in writing functions particular with large number of parameters where a good set of parameters may often take a default value to work with.

(Refer slide Time: 12:56)

The slide has a blue header bar with the title 'Default Parameter: Highlighted Points'. On the left, there is a vertical sidebar with a navigation menu:

- Module 08
- Partha Pratim Das
- Objectives & Outline
- Default Parameter** (highlighted in yellow)
- Function Overloading
- Overload Resolution
- Default Parameters in Overloading
- Summary

Below the sidebar, there is a video player showing a man speaking. The main content area contains a bulleted list of points:

- C++ allows programmer to assign default values to the function parameters
- Default values are specified while prototyping the function
- Default parameters are required while calling functions with fewer arguments or without any argument
- Better to use default value for less used parameters
- Default arguments may be expressions also

Handwritten notes in red ink are overlaid on the slide, showing examples of function calls:

val f(int = 2+3);
val f(int = 2+n);

We will highlight the basic points about default parameters, that is C++ allows programmers to assign default values to the function parameters, you have seen this. Default values are specified while prototyping the function. Prototyping the function,

that is what is meant is you write the default values in the function signature as we have seen. Default parameters are required while calling the function with fewer arguments or without any argument, so we have seen how one or two parameters as been used with their default value when they were missing in the call site.

As a practice, certainly it cannot be said as to which parameters should be default and which parameters should not be default. But as a practice it will be good to use default values for less used parameters, while you can actually use default value for all the parameters if you so desire. The default arguments, when you call the function the default arguments or even the default values can also be expression as long as they can be computed at the compilation time.

I can have a default values like $2 + 3$, as default parameters. This will be valid for a function, I can have a default value like $2+ 3$, what the compiler will do, it will compute $2 + 3$ while it is compiling and it will set the default value as 5. But please note that I cannot have a default value which is actually an expression at the time of compilation that is I cannot have something like, say, $2+n$, unless n is defined to be a constant value, constant integer. Because if n is variable then the compiler does not know at the time of compiling your program as what default value are you providing.

(Refer slide Time: 15:18)

The slide has a blue header bar with the title "Restrictions on default parameters". Below the header, there is a vertical sidebar containing a navigation menu with items such as "Module 08", "Partha Pratim Das", "Objectives & Outcome", "Default Parameter", "Function Overloading", "Overload Resolution", "Default Parameters in Overloading", and "Summary". The main content area contains a bulleted list of restrictions:

- All parameters to the right of a parameter with default argument must have default arguments (function f) —
- Default arguments cannot be re-defined (function g)
- All non-defaulted parameters needed in a call (call g())

Below the list, there is some C++ code with annotations:

```
#include <iostream>

void f(int, double = 0.0, char *); // Error C2640: 'f': missing default parameter for parameter 3
void g(int, double = 0, char * = NULL); // Error C2672: 'g': redefinition of default parameter : parameter 3
// Error C2672: 'g': redefinition of default parameter : parameter 2

int main() {
    int i = 5; double d = 1.2; char c = 'b';

    g(); // Error C2660: 'g': function does not take 0 arguments
    g(i);
    g(i, d);
    g(i, d, &c);
    return 0;
}
```

A red circle highlights the error message "Error C2640: 'f': missing default parameter for parameter 3" in the code. A red arrow points from the word "parameter" in the bullet point to this error message. There is also a large red "X" drawn over the entire code block.

Now, next we will try to see that is it that I can default the parameters of a function in any way that I desire, or there are some basic rules or restrictions that we will need to follow. So, we discuss some basic rules of a default parameters. The first rule is very important it says that all parameters to the right of a parameter with default argument must all have default arguments.

For this rule, look at this particular function. We are talking about function f which has three parameters, int, double, and pointer to character. In this case we have provided a default value to the second parameter, why we have not provided any default value to the third parameter which is char*. If you do that then the compiler will give you an error and, in this case, I have shown the error from one compiler, which says that f has missing default parameter for parameter three that is for this parameter.

So, it says that if I write any function and if I have one parameter, two parameter, three parameter, and if some parameter is defaulted then all parameters that I write after this default parameter all of them once this is the defaulted the next one has to be defaulted; the next one will also have to be defaulted all of them till the end we will have to be defaulted. Otherwise, compiler will not be able to resolve as to which parameters you are actually using. So that is the first rule, that once we default a parameter to it is right all parameters that exist must get default values

Second rule as in here is, default parameters cannot be redefined. You look at, say these two lines. Here, first in the first of this we have defined a function f which has three parameters again, int, double and char*, and two parameters have been defaulted the second parameter to 0 and third parameter to a null pointer, this is okay, naturally. Later on, we are again talking about that same function prototype g, but, if you look in to these two values, we are now using a different default value for the second parameter. So, this is not permitted, you can default the parameter value of a function only once.

Once you have defaulted then you cannot again specify the default parameter value. Now you think that this is the ambiguity, is from the fact that since you have defaulted double to zero and then you have defaulted double to one if you looking to this carefully that you have used two default values and that is the reason the compiler is giving this error

which is shown here as error on the parameter two, but incidentally compiler actually does not look at the actual value of the default that you have provided.

To understand this, please focus your attention to the default value of the third parameter. Here the third parameter was initially in the first case was given to be null and the second case it is redefined to be null again so you are actually using the same default value in the next time. But even then, the compiler gets confused and will tell you that, the default parameter for parameter three has been redefined, it was already defined and it is being defined again and this will not be allowed. It does not matter whether you are redefining a default value by the same value that you are defined earlier or you are redefining a default value using a different value that you are used earlier, in both cases this will not be permitted. So, default arguments cannot be redefined for functions where you are using the default parameters, this is the rule number two.

(Refer slide Time: 20:22)

Restrictions on default parameters

- All parameters to the right of a parameter with default argument must have default arguments (function f)
- Default arguments cannot be re-defined (function g)
- All non-defaulted parameters needed in a call (call g())

```
#include <iostream>

void f(int, double = 0.0, char *);  
// Error C2648: 'f': missing default parameter for parameter 3

void g(int, double = 0, char * = NULL); // OK  
void g(int, double = i, char * = NULL);  
// Error C2572: 'g': redefinition of default parameter : parameter 3  
// Error C2572: 'g': redefinition of default parameter : parameter 2

int main() {  
    int i = 5; double d = 1.2; char c = 'b';  
  
    g(); // Error C2660: 'g': function does not take 0 arguments  
    g(1); //  
    g(1, d, &c);  
    return 0;  
}
```

Annotations on the slide include:

- A red circle highlights the error message "Error C2648: 'f': missing default parameter for parameter 3".
- A red circle highlights the error message "Error C2572: 'g': redefinition of default parameter : parameter 3".
- A red circle highlights the error message "Error C2572: 'g': redefinition of default parameter : parameter 2".
- A red box encloses the line "g(); // Error C2660: 'g': function does not take 0 arguments".
- A red arrow points from the handwritten note "g(1, &c)" to the parameter "d" in the code.
- A red arrow points from the handwritten note "double = 0" to the parameter "d" in the code.

The third rule which says that since you are using default parameters you can choose to specify them at the time of a call or you may not specify them at the time of a call. So, if we look in the context of function g, let say this particular definition which is a certainly a correct definition then we can see that these three calls are valid, that is I can call the function with one int parameter g(i) which will be taken as a first or I can call it with the

first two parameters i and d which is int and double or I can call it with all the three parameters. All three of them are okay. Suppose, if you try to call the function g without any parameter then that will be an error, because if you have to call this function without any parameter then the first parameter should have been defaulted which in this case has not been done.

So, it says that if some parameters of a function are given default values then at least all the non-default parameters must be specified as actual parameters and the default parameters, in case of the default parameters you have a choice and have you can specify none of them or some of them, but if you specify some of them then you will always have to follow the rule that is, you will have to specify them in the left to right order. What I mean is suppose here, I am specifying if you look into this particular specification let me clean up and show again, suppose you are looking into this call, here the function has three parameters but we are specifying this two. So, the first of them i here is certainly non-default parameter which is mandatorily required and the second parameter d that you specify is taken to be double.

Now, suppose you tried to do a call like i, &c thinking that this call will mean that i will go for the int, this will go for the third parameter char* and the double value in the middle will be taken as 0.0. This will not be possible, this is an error. The reason is the compiler has no way to know that between the two default parameters double and char * which one you are specifying here and which one you are specifying here. What it has to assume that, if you are using these parameters then you can use the default values in a particular order that is, in the call the parameters actually given has to match from left to right it is only after a point you can skip all the parameters to the right of that formal parameter, but not skip something from the middle. So, these are some of the basic rules that the default parameters have to follow.

(Refer slide Time: 24:03)

The slide has a blue header with the title 'Restrictions on default parameters'. Below the header is a sidebar with navigation links: Module 08, Partha Pratim Das, Objectives & Outline, Default Parameter, Function Overloading, Overload Resolution, Default Parameters in Overloading Summary, and Help. The main content area contains a bullet point and several code snippets. A red annotation highlights the first bullet point: 'Default parameters should be supplied only in a header file and not in the definition of a function'. Another red annotation highlights the third parameter 'char ch' in the source code. A large red circle is drawn around the handwritten note 'g(int, double=0.0, char='a')' written on the right side of the slide.

- Default parameters should be supplied only in a header file and not in the definition of a function

```
// Header File: myFunc.h
void g(int, double, char = 'a');

// Source File: myFunc.cpp
#include <iostream>
using namespace std;
#include "myFunc.h"

void g(int i, double d, char c) {
    cout << i << ' ' << d << ' ' << c << endl;
}

// Application File: appa.cpp
#include <iostream>
#include "myFunc.h"
// void g(int, double, char = 'a');

void g(int i, double d = 0.0, char ch); // OK a new overload
void g(int i = 0, double d, char ch); // OK a new overload
int main() {
    int i = 6; double d = 1.2; char c = 'b';

    g();           // Prints: 0 0 a
    g(i);         // Prints: 6 0 a
    g(i, d);      // Prints: 6 1.2 a
    g(i, d, c);   // Prints: 6 1.2 b
}
```

Here, we show another; this is not exactly a rule by the definition of the language but I just show this as the way the practice of how you should be doing using the default parameters. So, he is saying that default parameters should be supplied only in a header file and not in the definition of the function, that is if you are doing a default parameter then this is as if you have a header file where you have specify a default this is the definition file the source file where you have defined the function it have provided the body for that function. So what we are saying is you should never specify any default parameter here or somewhere else where you are possibly using this function let say here.

Now, to understand let us see what is happening, this function has been define the prototype is given in the header and this is an application code this one is an application code which is using that function so it includes that header which means this particular function definition, this particular function prototype has been included. Now suppose in the body of your application you want to write something like this, surprisingly this all will be accepted these are all valid things. So, what you are saying here is, you are saying that from the header the third parameter was already defaulted, now you are saying my second parameter is also defaulted. so now I have; as if my function g, so these two together actually means that my function g has two of it's parameters defaulted and so

on.

Similarly, in the third case here we show that even the first parameter has been defaulted. The default parameter feature allows you to do this, but the difficulty is while you are using you will not know, which part of the default definition is in the header file and part of the default definitions are in your source file, implementation file. So, you will not know at any single point as to what parameters are defaulted and to which values they are defaulted. So, all defaults that are to be used; if you have to do something like this all of them should be moved to the header files so that at one point you can see what default parameters exist and what are their values, otherwise this mechanism of this structure of writing the code with default parameters can get really very confusing.

This is a restriction from practice point of view, language does allow it so if you look at the end of I have shown that with this definitions what you can achieve you can actually now call the function g with four different forms because all three parameters eventually have been defaulted, but it would always be better to default them at one place so that in one signature itself instead of defaulting them successively in three different signatures split between multiple files this will become very confusing for the purpose of use.

(Refer slide Time: 27:57)

Module 08
Partha Pratim Das
Objectives & Outline
Default Parameters
Function Overloading
Overload Resolution
Default Parameters in Overloading

Function overloads: Matrix Multiplication in C

- Similar functions with different data types & algorithms

```
typedef struct { int data[10][10]; } Mat; // 2D Matrix
typedef struct { int data[10]; } VecRow; // Row Vector
typedef struct { int data[10][1]; } VecCol; // Column Vector

void Multiply_M_R (Mat a, Mat b, Mat* c) { /* c = a * b */ }
void Multiply_R_VG (Mat a, VecCol b, VecCol* c) { /* c = a * b */ }
void Multiply_VR_M (VecRow a, Mat b, VecRow* c) { /* c = a * b */ }
void Multiply_VC_VR (VecCol a, VecRow b, Mat* c) { /* c = a * b */ }
void Multiply_VR_VC (VecRow a, VecCol b, Mat* c) { /* c = a * b */ }

int main() {
    Mat m1, m2, rm; VecRow rv, rrv; VecCol cv, rcv; int r;
    Multiply_M_R (&m1, &m2, &rm); // rm <-- m1 * m2
    Multiply_R_VG (&m1, cv, &rcv); // rcv <-- m1 * cv
    Multiply_VR_M (rv, &m2, &rrv); // rrv <-- rv * m2
    Multiply_VC_VR (&cv, &rrv, &rm); // rm <-- cv * rv
    Multiply_VR_VC (rv, &rcv, &rm); // r <-- rv * rcv
    return 0;
}
```

- 5 multiplication functions share same functionality but different argument types
- C treats them as 5 separate functions
- C++ has an elegant solution

NPTEL MOOCs Programming in C++ Partha Pratim Das 10

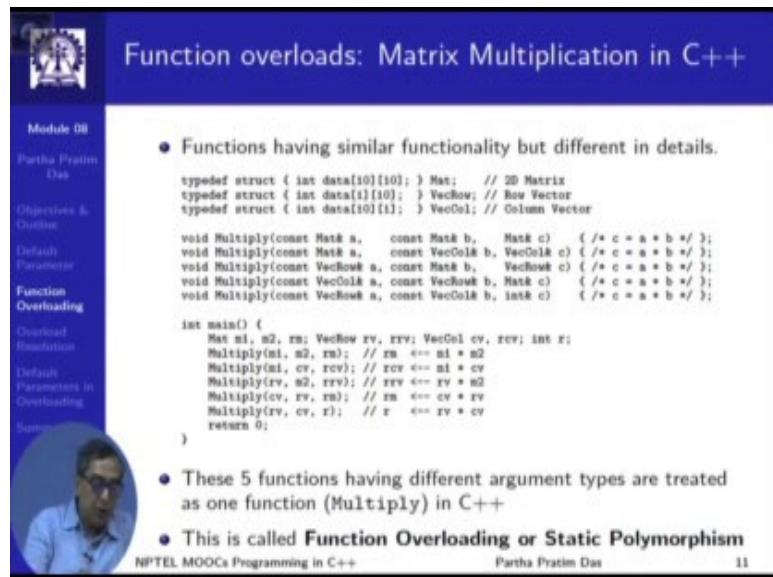
So, we have seen the default parameters feature of a C++ and we have seen how to use it and what are the restrictions for using that?

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 13
Default Parameters and Function Overloading (Contd.)

Welcome to Module 8 of Programming in C++. We have been discussing on this module in the earlier part and we talked about default parameters. Now, we will start discussing the function overloading.

(Refer Slide Time: 00:40)



Function overloads: Matrix Multiplication in C++

Module 08
Partha Pratim Das
Objectives & Outline
Default Parameters
Function Overloading
Default Parameters in Overloading
Summary

- Functions having similar functionality but different in details.

```
typedef struct { int data[10][10]; } Mat; // 2D Matrix
typedef struct { int data[1][10]; } VecRow; // Row Vector
typedef struct { int data[10][1]; } VecCol; // Column Vector

void Multiply(const Mat& a, const Mat& b, Mat& c) { /* c = a * b */ }
void Multiply(const Mat& a, const VecCol& b, VecCol& c) { /* c = a * b */ }
void Multiply(const VecRow& a, const Mat& b, VecRow& c) { /* c = a * b */ }
void Multiply(const VecCol& a, const VecRow& b, Mat& c) { /* c = a * b */ }
void Multiply(const VecRow& a, const VecCol& b, Mat& c) { /* c = a * b */ }

int main() {
    Mat m1, m2, rm; VecRow rv, rrv; VecCol cv, rcv; int r;
    Multiply(m1, m2, rm); // rm <= m1 * m2
    Multiply(m1, cv, rcv); // rcv <= m1 * cv
    Multiply(rv, m2, rrv); // rrv <= rv * m2
    Multiply(cv, rv, rm); // rm <= cv * rv
    Multiply(rv, cv, r); // r <= rv * cv
    return 0;
}
```

- These 5 functions having different argument types are treated as one function (**Multiply**) in C++
- This is called **Function Overloading or Static Polymorphism**

NPTEL MOOCs Programming in C++ Partha Pratim Das 11

To understand the function overloading, we will first take an example to illustrate why at all something like a function overloading is required and what it could possibly mean. So, here is an example in C. It is an example to multiply matrices, multiply vectors with matrices or multiply vectors themselves. So, if you look little carefully into the definitions given here, first we define three types, these are alias. Matrix is a two-dimensional square matrix that we have taken here in this example. Then, in the second VecRow. In this VecRow is a row vector and VecCol is a column vector. So, you can understand these. These you can look at this.

So, these are the three types and then, what we want is, wherever it is defined by the rules of matrix multiplication, we want to write function to multiply them. The first function multiplies two matrices and returns the result matrix. You can easily see that ‘Mat a’ is a square matrix a, ‘Mat b’ is another square matrix and both of them have size 10. So, if I multiply them, I will get another square matrix c as a result by the rules and you understand that since we are writing this in C, the call by value, we need to use pointers to get the output from this function.

Now, suppose I also want to define a multiplication which is the second one which is between a square matrix ‘a’ and a column vector ‘b’. If I do that, naturally I will get a column vector c. Now, if we can recap on the rules by which we multiply matrices, the row column mechanism which all of you know and I do not need to elaborate that here. You will know that there is hardly any difference between multiplying two matrices or multiplying a matrix or rather post multiplying a square matrix with a column vector of the same size. Only difference being your result will be a column vector and if you proceed, if we look into the third, it is a pre-multiplication of a matrix by a row vector.

If you look into the fourth, it is multiplication of a column vector by a row vector where the result turns to be a square matrix and in the fifth, you get ‘c’, the multiplication of a row vector by a column vector which turns out to be a single value. Now, all of this follows the same algorithm for multiplication, but if I want to express them, code them in C language, I will need to provide all different names. If you look into this part, I need to provide all different names to these functions because they are different functions. They take different kinds of, they all take three arguments because conceptually they are taking ‘a’ and ‘b’ matrices and multiplying and giving you a result ‘c’.

But since these are all different types, I need to have different names given to this function and when I am using, I not only need to understand what different parameters I am passing but corresponding to that i.e ‘rv’ is a row vector ‘m2’ is a matrix. So, I have to remember that, if I am using a row vector and a matrix, my name of the function has to be ‘multiply_VR_M’ or something similar. Whereas if I am doing multiplication of a row vector by a column vector, then the name of the function has to be something different e.g. Multiply_VR_VC.

So, if you summarize these observations, so what you understand is five multiplication functions that we show here, share the same functionality. They have same algorithm, but just they have five different kinds of argument types and result types and consequently C needs to treat them as five different separate functions. Now, C++ fortunately has an elegant solution for this situation, where different functions which have similar functionality maybe they use the same algorithm or maybe they use slightly different variations of the same algorithm, but certainly has to deal with different kinds of data types in the argument can now share their function name and can make the whole programming exercise lot easier.

So, to start with, continuing with the same example, now this is again coded in C++. Just look at the major differences. The first if you look into the `typedef`, they are identical. We are dealing with the same types. If you look into the five functions, we have the same five functions so far as the functionality is concerned that is the first one still multiplies two matrices, the second one multiplies a matrix, post multiplies a matrix with a column vector and so on, but if you look into the names, you see something very different. You see that all of them have the same name, something that you could not have done in C.

In C we formally say that function names are global. These are global symbols. So, in the whole of C program, irrespective of how many files in which you write and so on, but in the whole of the program that will run as one unit from call to one main, every function name has to be distinct, every function name. There can be only one function name for every different function that you write for the program, but here you can see that we have five different functions sharing the same name, but of course when you look into their parameters like in here, all the parameters you will find that no two of them have the same set of parameters.

They do not actually need to, because what distinguishes the first multiply which multiplies two matrices and gives through another square matrix from the second function which post multiplies a matrix with a column vector to give a column vector or the third one which pre-multiplies a matrix with a row vector to give a row vector. All of them have various different, their corresponding different data types for the arguments.

Now added with that the small thing that we have done is what we have learnt in module 6 and 7 that we have learnt to use the call by reference matrices at big objects. So, you would not like to pass them as value and copy them, rather we would like to pass them as reference and to make sure that the input matrices or vectors are not tempered within the function. We make those parameters constant whereas, on the output side we use just the call by reference mechanisms, so that we do not need that indirect pointer part, but this is just a matter of detail with the points to note is all these five have the same name and all these five can be used by the respective actual parameters with the same name.

So, what happens is suppose I am trying to do a post multiplication of a matrix with a column vector, so I have a matrix ‘m1’, I have a column vector ‘cv’ and if I multiply them, what should I get? I should get a column vector. So, let say I am expecting the column vector ‘rcv’ as a result. So, what I say I just write multiply; I put ‘m1’ as the matrix, ‘cv’ as a column vector and ‘rcv’ as my result matrix. Now, somehow this means that if I do this and I must call that particular function appropriately which multiplies; post multiplies a square matrix with a column vector.

So, very interestingly if you specify these parameters in this function, then interestingly this will actually call this function. Though there are five functions, all of them called as multiply and I am calling this function also as multiply, but somehow, I will be able to understand given the parameters, given the types of the parameters that I am actually interested in the second function with whom the types of these parameters, the type of this parameter is Mat which matches here, the type of the second parameter is parameter ‘cv’ is ‘VecCol’ which matches here and type of the third parameter rcv is again ‘VecCol’ which matches here. We find that this is the only function out of the 5, where these 3 parameters match in their type and the compiler would be accordingly able to call the second function here from this particular invocation.

Similarly, if I look into this, then depending on the types of these 3 parameters, the compiler would be able to actually call this function because ‘rv’ is ‘VecRow’, ‘rcv’ is ‘VecCol’ and ‘r’ is int. ‘rv’, ‘cv’, ‘r’, ‘VecRow’, ‘VecCol’, int will only specifically call the last function which is exactly what will work. So, these 5 functions have different argument types, but they are treated as one function having a common name in C++ and

this feature is very-very powerful, and which will enable us to do lot of things is called function overloading or as we will slowly understand an alternate name, more formal name for it is static polymorphism. So, we just saw the motivation of why the function overloading would be easy. We saw that in situation where we had to give five different function names to five different functions while their core functionality, core algorithm is still same and we had to remember all these names distinctly, we in C++ can use function overloading to just use the same name and depending on our call or use of the function, the appropriate function will get called appropriate function will be bound to the particular call we have seen that. So, now we go forward and see how we can do different kinds of function overloading in C++.

(Refer Slide Time: 13:27)

Same # of Parameters	Different # of Parameters
<pre>#include <iostream> using namespace std; int Add(int a, int b) { return (a + b); } double Add(double c, double d) { return (c + d); } int main() { int x = 6, y = 6, z; z = Add(x, y); // int Add(int, int) cout << "Int sum = " << z; double s = 3.6, t = 4.26, u; u = Add(s, t); // double Add(double, double) cout << "double sum = " << u << endl; } return 0;</pre>	<pre>#include <iostream> using namespace std; int Area(int a, int b) { return (a * b); } int Area(int c) { return (c * c); } int main(){ int x = 10, y = 12, z = 6, t; t = Area(x, y); // int Add(int, int) cout << "Area of Rectangle = " << t; int z = 6, u; u = Area(z); // int Add(int) cout << "Area of Square = " << u << endl; } return 0;</pre>
Area of Rectangle = 12 Area of Square = 36 • Same Add function • Same # of parameters but different types	

NPTEL MOOCs Programming in C++ Partha Pratim Das 12

So, in function overloading, we define multiple functions having the same name because unless you have more than one function having the same name, that issue of overloading does not arise and the second is the binding happens at compile type. Binding is a formal term to say that given the function call, how do you decide which particular function will actually be called and that process is known as binding. This is what the compiler does. So, this is called the binding at compile time.

(Refer Slide Time: 14:13)

Program 08.03/04: Function Overloading

Module 08
Partha Pratim Das
Objectives & Outline
Default Parameter
Function Overloading
Overload Resolution
Default Parameters in Overloading
Summary
Q&A

● Define multiple functions having the same name
● Binding happens at compile time

Same # of Parameters	Different # of Parameters
<pre>#include <iostream> using namespace std; int Add(int a, int b) { return (a + b); } double Add(double c, double d) { return (c + d); } int main() { int x = 5, y = 6, z; z = Add(x, y); // int Add(int, int) cout << "int sum = " << z; double s = 3.5, t = 4.25; u = Add(s, t); // double Add(double, double) cout << "double sum = " << u << endl; } return 0;</pre> <p>int sum = 11 double sum = 7.75 • Same Add function • Same # of parameters but different types</p>	<pre>#include <iostream> using namespace std; int Area(int a, int b) { return (a * b); } int Area(int c) { return (c * c); } int main(){ int x = 10, y = 12, z = 5, t; t = Area(x, y); // int Add(int, int) cout << "Area of Rectangle = " << t; int z = 5, u; u = Area(z); // int Area(int) cout << "Area of Square = " << u << endl; } return 0;</pre> <p>Area of Rectangle = 12 Area of Square = 25 • Same Area function • Different # of parameters</p>

Now, let us look into this. On the two columns, we will show two different kinds of instances here. We show the overloading of a function ‘Add’ where the first instance here, you just look at specifically focus on the parameters. The ‘Add’ takes two integer parameters and returns an integer in the second case, it takes two double parameters and returns a double. So, we are trying to write an ‘Add’ function which conceptually should add two numbers, but in C it does matter or in C++ also, it does matter as to whether the numbers it is adding are int or they are double or they are something else.

Now, I can write these two functions together in my same code and then, I am using it at two different places. Here I am using it `Add(x, y)`, where `x` is an int and `y` is an int. So, basically, I am calling the ‘Add’ with two int parameters. Therefore, this call will get bound to this definition of add which is what works for two int parameters whereas, if you look into the second call of the add function which uses ‘`s`’ and ‘`t`’ of type double, our parameter types are double and double the compiler will figure out that this call is actually for the second definition of the add function and will bind it there.

Accordingly, the first one will print a sum 11 and the second call will print a sum 7.75. As we all know that it does matter for addition as to what type of data you are using because if I am adding two integers, I have one kind of addition. If I am adding two

doubles, I have a different kind of addition. So, in this part we have seen that the function ‘Add’ has in both cases, the same number of parameters, but the types of the parameters are different and based on that we are able to resolve as to what a particular call is talking about, from the call, what particular function of the different alternatives that have been overloaded needs to be actually invoked, need to be actually bound. Now, it is not necessary for overloading.

So far, all the examples we have seen either the five multiply functions or the add functions here, the number of parameters have been same in all cases, but this is not mandatory for function overloading. So, look at the right column.

The next example, here we show two functions Area. The first one is meant to compute the area of a rectangle. So, it has to take two parameters of width and height, multiply them and written that, whereas, the second one takes only one parameter because it is supposed to compute the area of a square, so, you just take that and squares that number and written it. So, between these two functions, the name is same. This has two parameters whereas, this has one parameter and we will see below that we can still work with this given if you call ‘Area(x,y)’, then we know that there are one function. One of these function, this one has two parameters, the parameter x as here is int, parameter y is also int. So, it is a (int, int) call. So, two parameters, both of them should int. They are indeed int here. So, this call will actually invoke the first area function.

In contrast if we look into the second call, there is one parameter ‘z’ which is of type int. This will get bound to the second area function which has one parameter. So, appropriately different functions as overloaded, will get called even when there are different numbers of parameters. Note that though the numbers of parameters are different, in terms of the first parameter here the type is a same.

In the earlier cases, the number of parameters were same in the earlier cases. Just the different types were giving you the clue in terms of which is the right function to call, here the numbers are different even though the type is overlapping. You still are able to resolve, were still able to bind the right function for the right call. So, when you talk about function overloading, we are basically talking about situations where there is more

than one function which needs to be used and these functions are somewhat related in their functionality.

Now, certainly you will not overload a function to do square root with the function to sort ten numbers. I mean you could always call them; let say I use a name ‘myfunc’ and I overload ‘myfunc’, if the parameter is double, then it will find square root and if the parameter to ‘myfunc’ is an array then it would sort. It is according to function overloading rules, you will be able to do that, but certainly that will be a disaster use of this feature.

So, when we overload the basic conceptualization is that all overloaded functions must have very related functionality. They should be talking about very similar algorithms, but the types of the parameters that they use, number of parameters that they use for doing the computation has to be distinct, so that I can give the same name to all of these functions and that is mechanism by which function overloading works in C++

(Refer Slide Time: 21:16)

Module 08
Partha Pratim Das
Objectives & Outline
Default Parameters
Function Overloading
Overload Resolution
Default Parameters in Overloading
Summary

Program 08.05: Restrictions in Function Overloading

- Two functions having the same signature but different return types cannot be overloaded

```
#include <iostream>
using namespace std;

int Area(int a, int b) { return (a * b); }
double Area(int a, int b) { return (a * b); }
// Error C2668: 'double Area(int,int)': overloaded function differs only by return type
//           From 'int Area(int,int)'
// Error C2371: 'Area'! redefinition; different basic types

int main() {
    int x = 10, y = 12, z = 6, t;
    double f;

    t = Area(x, y);
    // Error C2668: 't=': unable to resolve function overload
    // Error C3801: 'Area': identifier not found
    cout << "Multiplication = " << t << endl;

    f = Area(y, z); // Errors C2668 and C3801 as above
    cout << "Multiplication = " << f << endl;
}

return 0;
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 13

There are some restrictions in terms of function overloading. Function overloading is primarily decided based on the signature of the function as we have discussed that in every time case, we are saying that, in C, how you resolve between two functions. They

must have different names, different functions. In C++, two functions or more than two functions can have the same name. How do you resolve? You resolve based on their signature, based on the number of parameters, based on the types of the parameters we have seen so many examples.

Now, the question is, if I have two functions which have; let us check this particular case here, I have two functions. Again area, both of them use the same number of parameters, both of them use the same types of parameters (int, int) and (int, int), but their return types are different. Now, please note such kind of functions cannot be used together. This is not possible in terms of function overloading.

So, perhaps this shown here, this is one typical error message from a compiler showing you that here it is saying, it takes the first function that is okay, when you tries to compile the second function when tries to look into that, it says, that is why it is talking about the second function which this double return type says overloaded function differs only by return type. It differs only by return type and that is not what is allowed and that is why this is considered to be a redefinition as if you are redefining the same function as if you would have given this if you would have compiled this example with C.

You would also find a very similar error message because this is a redefinition message. The first message is what is typical from the function overloading point of view that you cannot overload two functions unless their number of parameters differ or the number of parameters sustains, but still the types of the parameters are different at least for one parameter if their signatures are just different in the return type, then that overloading is not permitted.

(Refer Slide Time: 23:56)

The slide has a blue header bar with the title 'Function Overloading – Summary of Rules'. On the left, there is a vertical sidebar with a navigation menu. The menu items are: Module 08, Partha Pratim Das, Objectives & Outline, Default Parameter, Function Overloading (which is highlighted in blue), Overload Resolution, Default Parameters in Overloading, and Summary. At the bottom of the sidebar, it says 'NPTEL MOOCs Programming in C++'. The main content area contains a bulleted list of rules for function overloading:

- The same function name may be used in several definitions
- Functions with the same name must have different number of formal parameters and/or different types of formal parameters
- Function selection is based on the number and the types of the actual parameters at the places of invocation
- Function selection (Overload Resolution) is performed by the compiler
- Two functions having the same signature but different return types will result in a compilation error due to *attempt to re-declare*
- Overloading allows **Static Polymorphism**

At the bottom right of the slide, it says 'Partha Pratim Das' and '14'.

So, please keep this restriction in mind while you deal with, while you write overloaded functions. To sum up in terms of the rules that we have for function overloading will say that the same function name may be used in several definitions. That is a basic overloading concept. The function with the same name must have different number of formal parameters and /or different types of formal parameters we have seen examples of both. Function selection is based on the number and types of the actual parameters. We have seen examples here as well.

Then, this function selection as it is done by the compiler is known as overload resolution. What it means that, you have multiple functions overloaded. Multiple functions by the same name and you are given one call site; you are given one call where you are trying to figure out which of these multiple candidates should be used. That process is called process of resolution for the overloading, something that C++ compiler performs and, overtime, you will understand that it is actually extremely complex process and extremely powerful process and by that process, the compiler tries to decide which particular function of the given overloaded versions you are going to use or you intend to use in the call site, if the compiler feels to do that.

If two functions have the same signature but different return types or for some other reason the parameter types being different, but not distinct enough for the compiler to resolve. If compiler fails to do that, then the compiler will tell you that you are attempting to re-declare or you would tell that I am confused and will say that there is ambiguity in that and in such cases, you will have to do something different to make the function overloading work or avoid function overloading and to write the code in a different form.

Overloading is in a form gives you static polymorphism. We are not immediately going into depths of static polymorphism. We will talk about polymorphism later in the context of various C++ features. The core features here I would just like to explain the two words.

Polymorphism means, poly means- many and morph means- to change. So, polymorphism is accommodating multiple changes or multiple forms into one. So, here the polymorphism means that the same function by name, but it has depending on the parameter type, different behavior, different algorithms and different interfaces. So, you are trying to basically decide on this multiple form and then you say that this is static. What static here means that you want to do this all in the compile time that is the compiler should be able to decide between these multiple forms and tell you exactly which of these different forms, polymorphic forms you are trying to use at your function call. So, combined together this kind of decision or overload mechanism is known as static polymorphism.

Of course, there are other forms of polymorphism which are typically called dynamic or run type polymorphism which we will discuss after when we get into discussing the object base part of the C++ language.

Programming in C++

Prof. Partha Pratim Das

Department of Computer Science and Engineering Programming in C++
Indian Institute of Technology, Kharagpur

Lecture – 14 Default Parameters and Function Overloading (Contd.)

Welcome to Module 8 of Programming in C++. We have already discussed default parameters and we have discussed function overloading, the basic requirement of function overloading and how functions can be overloaded, that is, multiple functions can be there with the same name in C++. But, as long as they differ by their parameter type, number and types of parameters. And, we have observed that functions cannot be overloaded with the return type.

(Refer Slide Time: 00:54).

The slide has a dark blue header with the title 'Overload Resolution'. On the left, there's a vertical sidebar with a logo at the top and a list of navigation links: 'Module 08', 'Partha Pratim Das', 'Objectives & Outline', 'Default Parameter', 'Function Overloading', 'Overload Resolution' (which is highlighted in yellow), 'Default Parameters in Overloading', and 'Summary'. The main content area contains a bulleted list under the heading 'To resolve overloaded functions with one parameter':

- To resolve overloaded functions with one parameter
 - Identify the set of *Candidate Functions*
 - From the set of candidate functions identify the set of *Viable Functions*
 - Select the *Best viable function* through (*Order is important*)
 - Exact Match
 - Promotion
 - Standard type conversion
 - User defined type conversion

At the bottom of the slide, there are footer links: 'NPTEL MOOCs Programming in C++', 'Partha Pratim Das', and '15'.

Now, we would go next to give a glimpse of the overload resolution. Overload resolution is a process by which the compiler decides amongst multiple candidate functions, multiple candidate definitions of a overloaded function that exist. The compiler has to decide which particular one it should bind to. This is overload resolution, is a little bit advanced topic in function overloading. So, if you find it little difficult to handle at this

stage, you may keep it at bays and come back to it later when we discuss about the depths of overloading in C++, later on.

So, here we would just try to give you a basic idea that how to resolve the overloaded functions. I am using an example here with just one parameter to keep things simple. These are the basic steps which are to identify a set of candidate functions; that is, you first scan all functions that are been defined. Certainly, a function can be called, provided it has already been defined. So, the compiler scans all definitions that have been received so far, before this call and forms a set of candidate functions.

Then, it scans through the set of candidate functions that is this point scans to the set of candidate function to find a set of viable functions that match the number of parameters and so on. And then, it tries to do the most difficult part, which is, decides on what is the best viable function. And, for doing that it makes use of the following strategies. And, please note that the order is important here. That is, it applies the strategies, not arbitrarily. But, in this order of exact match, promotions, standard type conversion and user defined type conversion.

Now, certainly we cannot explain all of this right here. So, we will come back to it even at a much later stage. For now, I will just show you a simple example of how the resolution is done using exact match and talk a little bit about what is a scope of exact matching and what is the scope of promotion.

(Refer Slide Time: 03:28)

The slide is titled "Example: Overload Resolution with one parameter". It features a sidebar with a logo and navigation links for Module 08, including "Partha Pratim Das", "Objectives & Outline", "Default Parameter", "Function Overloading", "Overload Resolution", "Default Parameters in Overloading", and "Summary".

In the context of a list of function prototypes:

```
int g(double);           // F1
void f();                // F2
void f(int);             // F3
double h(void);          // F4
int g(char, int);        // F5
void f(double, double = 3.4); // F6
void h(int, double);     // F7
void f(char, char *);    // F8
```

The call site to resolve is:

```
f(5.6);
```

Resolution:

- Candidate functions (by name): F2, F3, F6, F8
- Viable functions (by # of parameters): F3, F6
- Best viable function (by type double – Exact Match): F6

NPTEL MOOCs Programming in C++ Partha Pratim Das 18

So, let us take an example. Let us take an example of overload resolution with one parameter. Let us assume; concentrate on the list on top here. Let us concentrate on this list. Let us assume that our target actually is to find out, for this particular call of a function by name *f* and parameter 5.6 to decide, which of the signatures this function corresponds.

That is, if I have said *f* (5.6), if the user has said that then which particular function has the user intended to call. So, in the context we are taking that before this call of *f*(5.6) has been encountered 8 different function headers or function definition has been seen by the compiler. And, just to refer to them conveniently we have called them as F1 to F8. Now, how do you decide from? So, certainly call of *f*(5.6) has to match exactly with one of them, so that I can say that a particular function is getting called. So, what are the steps in resolution? if you look at here, if you use steps in resolutions, first you form the candidate function.

In this case, the simple way to form the candidate function is to look for the name and find functions, which has a matching name. Here, we do not have any other factors. So, we will just go by name; the name is *f* here. So, let me look at what are the functions

which have name f; this function, this function, this function, this function. So F2, F3, F6, and F8 are my candidate function.

Certainly, if I say $f(5.6)$, I could not have meant this function g, even though g has one parameter, even though g has a parameter type which is double, which is the type of 5.6. But, certainly I could not have meant that because my names are different. So, my candidate functions give me these four functions. So, now, I focus only on this candidate function.

Now from this, based on this candidate function next screening that I do is based on the number of parameters. So, out of these candidate functions, I see what are the number of parameters. So, if I look at F2, this is 0; if I look at F3, it is 1; if I look at F6, it is 2 or 1 and if I look at this, this is 2. So, these are the required number of parameters for calling the respective candidate functions. And, how many parameters we have here? 5.6 - one parameter.

Second, immediately say that this is ruled out. I cannot call F2 by $f(5.6)$. I cannot similarly call this because it requires two parameters. So, my choice boils down to these two; it is either F3 or F6. So, I call them the set of viable functions.

Now between F3 and F6, I have to decide which one to use. So, I can think of that I am actually calling the function F3, which needs an int parameter. So, if I have to call function F3, then what is to happen? It has to happen that this parameter actually is a double parameter; 5.6 is a double. So, if have to call it F3, then this double has to get converted to int; which means, some truncation have to happen. And actually, the call will turn out to be F5. And then, I will be able to call it. Otherwise, I can call F6, where in, it is $f(5.6$ as a first parameter, second parameter is defaulted). So, this could be the call $\{f(5.6, 3.4)\}$. So, either this is the call or this is the call.

Now, the exact match say that I will choose this because here the type of the parameter, formal parameter and the type of the actual parameter, these two matches. Both of them are double. Whereas between this and this, I need to convert, one is double. I need to convert, actually truncate that to maintain. So, that is not a preferred one. So, the exact

match strategy, by using the type double tells me that my resolved function is F6. So, once I am done, the compiler will resolve that you have called this function.

This is the process of overload resolution. And, it is a, quite a complex process because so many different types could be involved. So, many different cases can happen. And, so while somebody who writes a compiler needs to know it very thoroughly. As a programmer, it is good to have some idea because you may have meant certain overload to be used. If the compiler did uses a different kind of overload, then you are using a wrong function at a right place. So, just let us take a quick look into some of these strategies. Exact match is the first one. That just to recap, these are the different ways to resolve the best function, exact match promotion and so on. So, you just are looking at what is an exact match.

(Refer Slide Time: 09:51).

The slide has a dark blue header bar with the title "Overload Resolution: Exact Match". Below the header is a sidebar containing a logo, the text "Module 08", "Partha Pratim Das", and a vertical list of topics: "Objectives & Outline", "Default Parameter", "Function Overloading", "Overload Resolution", "Default Parameters in Overloading", and "Summary". To the right of the sidebar is a main content area with a bulleted list of conversion types:

- lvalue-to-rvalue conversion
 - Most common
- Array-to-pointer conversion
 - Definitions: `int ar[10];`
 - `void f(int *a);`
 - Call: `f(ar)`
- Function-to-pointer conversion
 - Definitions: `typedef int (*fp) (int);`
 - `void f(int, fp);`
 - `int g(int);`
 - Call: `f(5, g)`
- Qualification conversion
 - Converting pointer (only) to const pointer

At the bottom of the slide, there is a footer bar with the text "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and the number "16".

So, exact match is; these are different cases of exact match. That is, you can convert an lvalue to an rvalue; that is, if you are using a variable just in case you are not familiar with this terms, l value is left value or basically the address of the variable. And, rvalue is a value; is a right value. So, it is like if I am writing `a=b` (`a` assign `b`), then what I am interested in `b` is its value of the `b`. and, what I am interested in `a` is the location where `a`

exist where I can go and keep the value of b. So, here I am interested in the lvalue and here I am interested in the rvalue.

So, you may need to convert from lvalue to rvalue. The other possibilities are like this, where you convert and you are trying to pass an array, but the actual function needed a pointer. You can convert arrays seamlessly to pointer. You can convert function to pointer. For example, this is the function pointer. We have seen what function pointers are. This is a function that I am trying to resolve. Which, second parameter is a function pointer; this particular function pointer. But, what I have passed is actually a function. I did not pass a pointer. So, I should have passed &g, which is a pointer.

But, I have just passed the function name directly. So, this conversion is allowed as a match. And, I can convert pointers to constant pointers. So, these are different cases of exact match that the overload resolution uses. Besides that, you could take promotions and conversions.

(Refer Slide Time: 12:05)

Overload Resolution: Promotion & Conversion

Module 08
Partha Pratim Das
Objectives & Outline
Default Parameter
Function Overloading
Overload Resolution
Default Parameters in Overloading
Summary

- Examples of Promotion
 - char to int; float to double
 - enum to int / short / unsigned int / ...
 - bool to int
- Examples of Standard Conversion
 - integral conversion
 - floating point conversion
 - floating point to integral conversion

The above 3 may be dangerous!

 - pointer conversion
 - bool conversion

NPTEL MOOCs Programming in C++ Partha Pratim Das 17

For example, I can convert character to integer, enum to integer, bool to integer. Or, I can convert between other kinds of integral conversions float to double, double to float; all of that various kinds, various types of a pointers can be converted. So, all these promotions

and conversions are also used for overload resolution. This is just to give you the glimpse. We will come back to this much later when we talk about variety of other overloading that are possible in C++. So, this is just to give you a basic idea.

(Refer Slide Time: 12:43).

The slide has a blue header with the title "Example: Overload Resolution fails". On the left, there's a sidebar with navigation links: Module 08, Partha Pratim Das, Objectives & Outline, Default Parameter Function Overloading, Overload Resolution, Default Parameters in Overloading, and Summary. Below the sidebar is a video player showing a man speaking. The main content area contains the following text and code:

- Consider the overloaded function signatures:

```
int fun(float a) {...}           // Function 1
int fun(float a, int b) {...}     // Function 2
int fun(float x, int y = 5) {...} // Function 3

int main() {
    float p = 4.5, t = 10.5;
    int s = 30;

    fun(p, s); // CALL - 1
    fun(t);    // CALL - 2
    return 0;
}
```

- CALL - 1: Matches Function 2 & Function 3
- CALL - 2: Matches Function 1 & Function 3
- Results in ambiguity

At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" with a profile picture, and the number "19".

Now, I will end. Before I end, I will also show that an overload resolution may fail in different cases. And, you will have to be careful about those. For example, here look at these three functions; the Function 1 has one float, Function 2 has two parameters, Function 2 also has two parameters with one defaulted. And, this is the use that you are trying to do. So, if you try to resolve at this, first CALL - 1, which is taking two parameters, then what are your candidates are? All three functions are your candidates; Function 1, 2, 3, because the name matches.

Now, what is viable? Viable is looking at the number of parameters. It is Function 2 and Function 3. Now between these two, Function 2 and function 3, what is the best one? It is difficult to say what is the best one because p is a float here, which matches the first parameter type of both these functions. If we look into the second, which is s, is int, which matches the second parameter type of both these functions. So, it is not possible for you to resolve between function 2 and function 3. So, if we have such a call, then the compiler will come back and tell you that, it is ambiguous. The compiler will say, "I am

confused, I do not know what to do". So, these are different cases, where resolution overload resolution will fail.

The second one is another case, where we are just using one parameter for function. So, naturally your candidates again are all three. But, your viable one, certainly two does not remain viable because here you are using just one parameter. Function 2 would have needed two parameters. So, both of these become viable. But again, if both of these are viable, you cannot easily resolve between them because you would, this could call this by a conversion, by a promotion of int to float; we just seen. Or, it could call 3 by assuming this is defaulted.

And, this is another promotion again from int to float. So, both of them has same kind of, you know, efforts, same kind of complexity. And therefore, this will result in ambiguity. So, whenever you write overloaded functions, you will have to be careful that such cases are not there in your code or if the compiler refuses to compile this and says that there is an ambiguity, please look for some of these situations that may exist in your code right now.

(Refer Slide Time: 15:40)

Program 08.06/07:
Default Parameter & Function Overload

Module 08
Partha Pratim Das

Objectives & Outline
Default Parameter
Function Overloading
Overload Resolution
Default Parameters in Overloading
Summary

• Compilers deal with default parameters as a special case of function overloading

Default Parameters	Function Overload
<pre>#include <iostream> using namespace std; int f(int a = 1, int b = 2); int main() { int x = 5, y = 6; f(); // a = 1, b = 2 f(x); // a = x = 5, b = 2 f(x, y); // a = x = 5, b = y = 6 return 0; }</pre> <ul style="list-style-type: none">• Function f has 2 parameters overloaded• f can have 3 possible forms of call	<pre>#include <iostream> using namespace std; int f(); int f(int); int f(int, int); int main() { int x = 5, y = 6; f(); // int f() f(x); // int f(int) f(x, y); // int f(int, int) return 0; }</pre> <ul style="list-style-type: none">• Function f is overloaded with up to 3 parameters• f can have 3 possible forms of call• No overload here use default parameters

NPTEL MOOCs Programming in C++ Partha Pratim Das 20

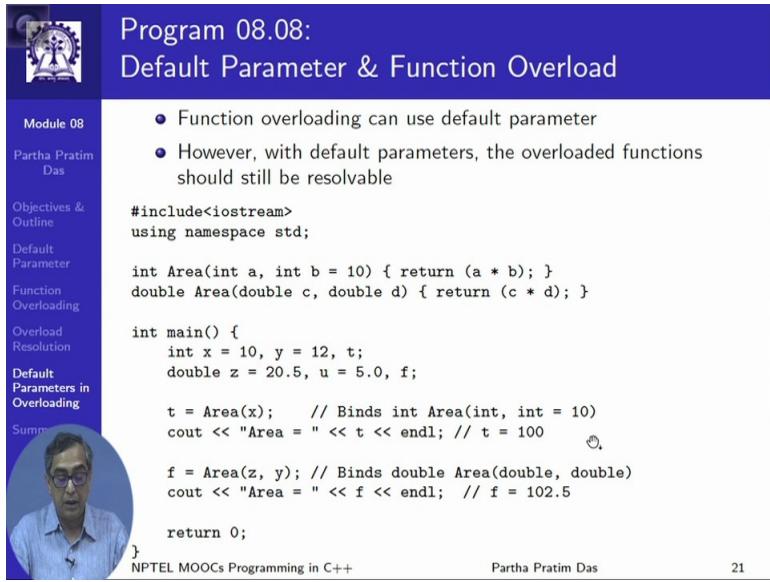
Now, we have looked at default parameters and we have looked at function overloading. Now, you can just observe that the reason we have put them together in the same module is the fact that basically they are; at the core they are basically the same feature, except for maintaining that default parameter value, in case of the default parameters. The whole mechanism in the language is the same. So, if we look at these two codes side by side from left and right columns here and if you look at the main function, the main function here is identical. It has, this is exactly the same code on both of these. here we have one function with two parameters defaulted and here we have three functions overloaded.

Now in terms of call, all these three calls resolve to the same function. Except that, it depends on how many parameters you are using. And here, these resolve to the respective functions, depending on this is the simple resolution; because in each one of these case, as you can figure out there only be, all three will be in the candidate set. And, in every case there will be only one function in the viable set. So, there is nothing to look for.

So, basically what you can think of that if you default a set of, the set of parameters in the function, then you always are creating two choices for a default parameter that either it may exist there in the call site or it may not exist in the call site. Depending on that, you are as if generating two different signatures for the same function. So, what I am saying is, if I write this function; that means, I can call int with int int, which basically is this overload. It also means that I can skip the second parameter. So, I can call it with just one int, which is this overload.

It also means that I may not specify any parameter. Just call it without parameter, which is this overload. So, underline, default parameter is basically nothing other than a special case of function overloading. With the added feature that it also carries the overload value, which is the initial, the value of the parameter along with it. Otherwise, depending on the function site, call site like this, or like this, which function will be bound is again a overload resolution problem.

(Refer Slide Time: 18:31)



The slide is titled "Program 08.08: Default Parameter & Function Overload". It features a sidebar with navigation links: Module 08, Partha Pratim Das, Objectives & Outline, Default Parameter Function Overloading, Overload Resolution, Default Parameters in Overloading, and Summary. A circular profile picture of Partha Pratim Das is also present. The main content area contains two bullet points under the heading "Function overloading can use default parameter":

- Function overloading can use default parameter
- However, with default parameters, the overloaded functions should still be resolvable

```
#include<iostream>
using namespace std;

int Area(int a, int b = 10) { return (a * b); }
double Area(double c, double d) { return (c * d); }

int main()
{
    int x = 10, y = 12, t;
    double z = 20.5, u = 5.0, f;

    t = Area(x);      // Binds int Area(int, int = 10)
    cout << "Area = " << t << endl; // t = 100

    f = Area(z, y); // Binds double Area(double, double)
    cout << "Area = " << f << endl; // f = 102.5

    return 0;
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 21

So, now naturally you can use default parameters with function overloading. Some of the earlier examples, I have already used that. But, here I am just explicitly wanted to discuss it that I can, between the overloaded functions themselves, some parameters could be default. But, as long as you can resolve the whole set of functions with the default parameters from the call site, these all usages are fine.

So, here is an example again here. So, the area function, we had this function, we had seen earlier. So, here both of them have two parameters. They are different types. And, in this case one parameter is defaulted. So, if we make this call, since, only this function can be called with one parameter. This will get resolved here. If we call this, then z is of type double and y is of type int. So, you will not be able to call the first function. We will call the second function because the first parameter matches as double. And, it is easier to convert an int to a double than converting a double to an int because if you convert a double to a int, you lose information.

If you convert int to double, then you do not lose information. So, that is the promotion strategy which will tell you that the bindings will happen, as this will bind here and this will bind here. That is what we have shown at this point. So, bottom line is what we are

trying to illustrate is default parameters and function overloading can be mixed together as long as the whole thing can be resolved. And, of course it can fail.

(Refer Slide Time: 20:25)

The slide is titled "Program 08.09: Default Parameter & Function Overload". It features a sidebar with a profile picture of Partha Pratim Das and a navigation menu for Module 08, including "Objectives & Outline", "Default Parameter", "Function Overloading", "Overload Resolution", and "Default Parameters in Overloading". The main content area contains a bullet point and a code snippet:

- Function overloading with default parameters may fail

```
#include <iostream>
using namespace std;
int f();
int f(int = 0);
int f(int, int);

int main() {
    int x = 5, y = 6;

    f();      // Error C2668: 'f': ambiguous call to overloaded function
              // More than one instance of overloaded function "f"
              // matches the argument list:
              //   function "f()"
              //   function "f(int = 0)"

    f(x);    // int f(int);
    f(x, y); // int f(int, int);

    return 0;
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 22

For example, this is, again in some way we had seen this earlier. But again, just to specifically note that here are three functions which are overloaded; one this is a zero parameter, this is one parameter and this is two parameters. So, given that these three calls are certainly resolvable. It will, that in every case of the viable function set will have only one function each; will be able to do that. But, the moment you put a default parameter to one of this overloaded function, you have a problem because now this function can have one parameter or zero parameter.

So, when I write a function call with zero parameter, your viable set will have both of these in place. And, certainly since there are no parameters at all. There is no match that you can try out because if there is no parameter, then what type you are going to match between the type of actual parameter and the type of the formal parameter. And therefore, the compiler cannot resolve. And, this is again from a typical compiler. This is the kind of error message that you will get. And, just see what the compiler is saying; is ambiguous call to overloaded function. And, it says that it cannot resolve between these two functions.

So, while you will use default parameters with overload, will have to remember that you cannot only look the overloading, nor you can just look at the default parameter. You have to look at the with default parameters, what are the different overloads that are getting created and whether those different overloads, whether they can be actually resolved by the outline of the mechanism that we have explained.

(Refer Slide Time: 22:17)

The slide has a dark blue header with the title "Module Summary". On the left, there's a sidebar with a logo and navigation links: "Module 08", "Partha Pratim Das", "Objectives & Outline", "Default Parameter", "Function Overloading", "Overload Resolution", "Default Parameters in Overloading", and "Summary". The main content area contains a bulleted list of topics covered in the module:

- Introduced the notion of Default parameters and discussed several examples
- Identified the necessity of function overloading
- Introduced static Polymorphism and discussed examples and restrictions
- Discussed an outline for Overload resolution
- Discussed the mix of default Parameters and function overloading

At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" with slide number "23".

So, this is; these are the features. So, at the end I would just like to sum up that in this module we have tried to address a major aspect of C++ language that is a function overloading. First, we have taken in look at the notion of default parameters, which later on we have explained, is a special case of function overloading. Default parameters make it very easy to write functions with large number of parameters to provide their default values and make the function libraries easier to use. And, function overloading is another feature which allows us to write multiple functions with the same function name, as long as their parameter types differ.

So, we had looked into variety of nuances in this regard. And, we have talked of basic static polymorphism. And most importantly, we have also discussed the mechanism by which, I mean; we have discussed the outline for the mechanism by which different functions can be resolved for the overload.

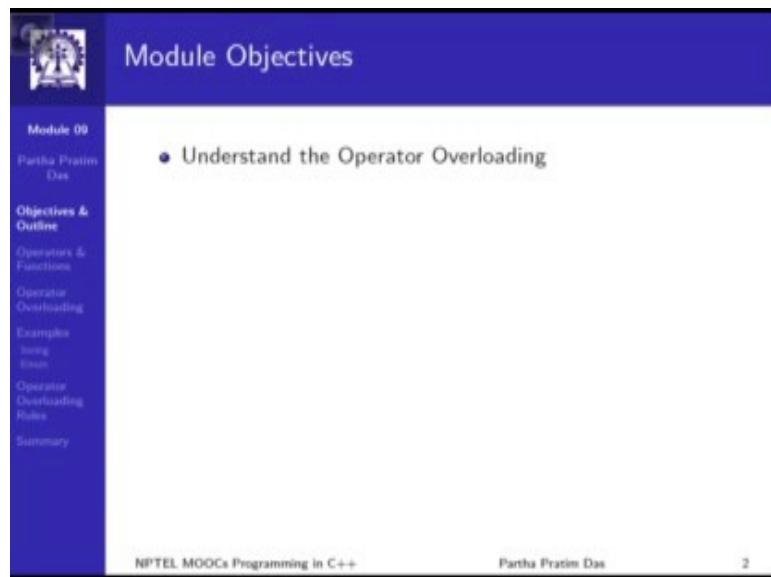
With this we will end this module. And, in the next one we will extend the same concept. And, see that similar overloading concept can be applied in case of operators also in C++, which will be known as operator overloading.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 15
Operator Overloading

Welcome to module 9 of programming in C++. We are continuing the discussions on the procedural extensions of C for better C in C++. We have already discussed a couple of features like const, inline function, reference and particularly function overloading and default parameters.

(Refer Slide Time: 01:13)



The slide has a blue header bar with the title "Module Objectives". On the left, there is a sidebar with a logo and navigation links for "Module 09" and "Objectives & Outline". The main content area contains a bulleted list: "Understand the Operator Overloading". At the bottom, it shows "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" along with a page number "2".

We will extend on the discussions on function overloading into a new topic, operator overloading. We will see, this is a new concept in C++, where new behaviors of operators can be defined, and we will take a look how that can be done in C++.

(Refer Slide Time: 01:19)

This slide is titled 'Module Outline' and is part of 'Module 09' by Partha Pratim Das. The slide content includes a list of topics: Basic Differences between Operators & Functions, Operator Overloading, Examples of Operator Overloading (including operator+ for String & Enum), and Operator Overloading Rules. A video thumbnail of the instructor, Partha Pratim Das, is visible on the left.

Module 09
Partha Pratim Das

Objectives & Outline
Operators & Functions
Operator Overloading
Examples
String Class
Operator Overloading Rules
Summary

Module Outline

- Basic Differences between Operators & Functions
- Operator Overloading
- Examples of Operator Overloading
 - operator+ for String & Enum
- Operator Overloading Rules

NPTEL MOOCs Programming in C++ Partha Pratim Das 3

So, the objective of this module would be to understand operator overloading. This is how we will do this, we will try to understand operators and functions better, and then define operator overloading take examples and set down the rules for it.

(Refer Slide Time: 01:35)

This slide is titled 'Operator & Function' and is part of 'Module 09' by Partha Pratim Das. It compares operator and function multiplication. The slide shows a C++ code snippet where a function 'Multiply' is used instead of the '*' operator. Both result in the same computation (c = a * b).

Module 09
Partha Pratim Das

Objectives & Outline
Operators & Functions
Operator Overloading
Examples
String Class
Operator Overloading Rules
Summary

Operator & Function

- What is the difference between an operator & a function?

```
unsigned int Multiply(unsigned x, unsigned y) {  
    int prod = 0;  
    while (y-- > 0) prod += x;  
    return prod;  
}  
  
int main() {  
    unsigned int a = 2, b = 3;  
  
    // Computed by '*' operator  
    unsigned int c = a * b; // c is 6  
  
    // Computed by Multiply function  
    unsigned int d = Multiply(a, b); // d is 6  
  
    return 0; //  
}
```

- Same computation by an operator and a function

NPTEL MOOCs Programming in C++ Partha Pratim Das 4

We start with asking a question, the question is we know in the programming language we often use operators, if we just look into this code, there is an operator here; a and b are two integer variables, we use the operator multiply which in this case a binary operator, which means it takes two operands, these operands are, a and b. They will operate on this operand and will give us a result. Similarly, if you look at here, I have function by the name Multiply, which also takes two arguments, two parameters a, and b; and using this algorithm, since this is a function I have to specify the definition of the function.

I have specified the algorithm by this algorithm it does multiply the first operand and second operand together and gives us a result. So, if you ask the broad question, the abstract question as to what the difference between an operator and a function is, you will find some very significant similarity, because both the operator as well as the function takes some parameters and compute the result. Multiply operator, in this case, takes two parameters, two arguments or often we said two operands and produce as a result.

Similarly, here I have a multiply function, which takes two operands or two parameters, in case of function we often say it is arguments or parameters and returns a result value. So, we see that there is a significant similarity if we look at the operators and function from the computational point of view. But still we choose to call some of the cases as operator, $a+b$, we say + (plus) is an operator, $a * b$ (a into b)- star is an operator, $-a$ ($minus a$), we say negation is an operator; whereas square root, sort, multiply function, all these we say are functions. So, our basic question we are asking is, what is the difference between an operator and function.

(Refer Slide Time: 04:14)

Difference between Operator & Functions	
Operator	Function
<ul style="list-style-type: none">• Usually written in infix notation• Examples: Infix: $a + b$; $a ? b : c$; Prefix: $++a$; Postfix: $a++$;• Operates on one or more operands, typically up to 3 (Unary, Binary or Ternary)• Produces one result• Order of operations is decided by precedence and associativity• Operators are pre-defined	<ul style="list-style-type: none">• Always written in prefix notation• Examples: Prefix: $\max(a, b)$; $qsort(int[], int, int,$ $\quad \quad \quad void (*) (void*, void*))$;• Operates on zero or more arguments• Produces up to one result• Order of application is decided by depth of nesting• Functions can be defined as needed

So, here we chart out the various similarities and differences between an operator and a function. The notable of this is a very first point, you see an operator is always written in the terms of an infix notation, this is how an operator written. So, the operator symbols it is in the middle and the two operands this binary operator, so it takes two operands the two operands sits on its two sides. Consider the case of this operator, these are written little bit differently in some cases the operator precedes the operand; we say this is a prefix case. In some cases, the operator follows the operand, we say this is a post fix case, but most often we will write things like $a * b$ (a into b), we will write a/b (a divided by b), $a-b$ (a minus b), $a\&\&b$ (a ampersand-ampersand b), and so on and so forth, minus a , all different once. But all of these are necessarily in the infix notation.

Now, if you look into the function, you all know, what is a function suppose this is a function or I have an another; I write in another function $\text{add}(a, b)$ will always; see that the function name occurs before the list of parameters that the function takes. So, a function is always written in the prefix notation. So, primarily what do we say is an operator, what do we say is a function has to is primarily first decided by the notation that we use for it and computationally, they are interchangeable; computationally where have an operator maybe I could have had a function or where have a function I may be able to have an operator and so on. But notation wise it is very different function is

necessarily in prefix notation, operators are usually in infix notation. But some of the operators in the infix also could be a prefix position or post fix positions.

Then the following points tell us some of the more commonly observed differences operators operate on one, two or three operands. We know in C, we have unary, binary and ternary operators. Operators having more than three operands are not usual, whereas a function can have any number of arguments. In fact, a function may not have an argument also, where it is never possible that I have an operator which does not have operate and a function can have 3, 4, 5, 10 any number of arguments. Coming to the third point, an operator will always produce a result, it is a part of an expression that will always produce a result where is the function usually produces a result, but we you already know that I can have a function with void return type which may not produce a result.

(Refer Slide Time: 07:36)

Operator	Function
<ul style="list-style-type: none"> Usually written in infix notation Examples: Infix: $a + b$; $a \neq b : c$; Prefix: $++a$; Postfix: $a++$; Operates on one or more operands, typically up to 3 (Unary, Binary or Ternary) Produces one result Order of operations is decided by precedence and associativity Operators are pre-defined $a+b*c$ 	<ul style="list-style-type: none"> Always written in prefix notation Examples: Prefix: $\max(a, b)$; $\sqrt{int[]}$, int, int, $void (*) (void*, void*)$; Operates on zero or more arguments Produces up to one result Order of application is decided by depth of nesting $f(g(x, y), z)$ Functions can be defined as needed

If I have an expression, in which say I have an expression like $a + b * c$. If we have an expression, we know that the order in which the different operators in this expression are evaluated or applied depends on the precedence and associativity of the operators. Whereas in case of function, it necessarily depends on if I have a function say like this, if I have function called like this. I will necessarily know that, this function call for g which

takes b and c and computes a result has to precede the function call to f. So, the order of application of function is dependent, is decided only by the depth of nesting all parameters have to get evaluated before the function can be applied.

Finally, operators are necessarily predefined, functions can be defined as and when needed. So, these are some of the major differences between the operator and the function, but it can be noted that for limited number of operands. And when always there is a result, it may be very convenient to be able to express computations in terms of operators, because as we say the operators allow us write algebra. For anything that we want to do for int, we can write a big expression for double, we can write such expressions, and we can write similar expressions for say pointer types and so on. It would be good if we could do similar things for other kinds of types as well.

(Refer Slide Time: 09:30)

Operator Functions in C++

- Introduces a new keyword: `operator`
- Every operator is associated with an operator function that defines its behavior

Operator Expression	Operator Function
<code>a + b</code>	<code>operator+(a, b)</code>
<code>a = b</code>	<code>operator=(a, b)</code>
<code>c = a + b</code>	<code>operator=(c, operator+(a, b))</code>

- Operator functions are implicit for predefined operators of built-in types and cannot be redefined
- An operator function may have a signature as:

```

MyType a, b; // An enum or struct
MyType operator+(MyType, MyType); // Operator function
a + b // Calls operator+(a, b)

```

- C++ allows users to define an operator function and overload it

NPTEL MOOCs Programming in C++ Partha Pratim Das 6

Next what we look at and what C++ introduce us is the notion of operator functions. That is if I have an operator then, so far the operators, that we have seen like operator plus for int or operator plus for double or operator multiplication for int. The behavior of this operators, that is the algorithm that the operator follows and the kind of result that the operator returns; is pre-decided by the compiler. Now, we are saying that to we want to do something different, for an operator we will associate a function with that operator

and to do that association C++ introduces a new keyword called ‘operator’, this is how this is spelt it is a new keyword. It is the reserved word as well and with this, we make some notational equivalence between the infix notation of the operator and the corresponding operator function that may be thought of.

So, if we have an expression $a + b$ which is very common as you will see; I can think of this corresponding to operator plus, I can think of, there is a function whose name is operator the plus symbol. So, given an operator the corresponding operator function gets a name which is a operator keyword followed by the operator symbol. So, $a + b$, the plus operator gets the name operator plus, there are two operands here a , and b naturally they become the two arguments a , and b in that same order. If you look at a different case, let say $a = b$ (a is assigned b) you would recall that in C assignment ($=$) is actually an operator, $a = b$ (a assigned b) is actually an expression, which has a value which is a value assigned to the left hand side.

So, if assignment is an operator then for this assignment symbol, I can write a corresponding operator function operator assignment and pass a and b as two parameters, whenever we write operator function it is very important to remember that the order of the operands, if there are more than one operands, the order of the operands cannot be changed. The first operand has to be become the first argument in the operator function and so on. So, in the same way this is just the third example is just to elaborate little bit more, that if I have $c = a + b$ (c assigned $a + b$) then, we know that plus has a higher precedence over assignment; so first there addition has to happen and then this added value will be assigned. So, in terms of operator function notation, this is what $\text{operator}+(a, b)$ (operator plus a b) is what adds and by nesting, this will necessarily happen earlier, we have already discussed in terms of functions the order is decided by nesting. So, this is more nested. So, this will happen first and the result that it produces will be operated with operator assignment that is it will be assigned to c .

So, corresponding to any operator-based expression that we have, we can write equivalent expressions using operator function notation. Please note at this point, that the operator functions are implicit for all predefined operators of built in types. That is all the examples, that I am showing above, in that a , and b these operands cannot be assumed to

be of int or double or float type, because for them the operator function notation has not been allowed. Because we do not expect that you would like to define a new kind of addition for integers. So, to come to how you do it, how you take a look at the operator function, say MyType is, some type which I am just assuming that it could be an enum type or a struct type as I said it cannot be a built-in type. So, it is some type that, I am defining and will see more lot more of these as we actually going C++. We actually have class definitions coming in. So, my type is some type defined, a, and b are two variables.

So, for in that context, say I can define an operator+ (operator plus) for MyType, which takes two MyType values, because we know operator plus is; a binary operator. So, it needs two values, so it has two arguments and it returns a MyType value as a result, and this is what we says the operator function for operator plus, that we can define; an if this has been defined the interesting thing is then, for this MyType value, I can write a+b as an expression. If I write a+b, then this expression actually calls the operator + (a, b), that is the operator function that have written for operator plus for my particular class. This is the very, very interesting part and this is what we will particularly focus on here.

(Refer Slide Time: 15:30)

Concatenation by string functions	Concatenation operator
<pre>#include <iostream> #include <cstring> using namespace std; typedef struct _String { char *str; } String; int main() { String fName, lName, name; fName.str = strdup("Partha "); lName.str = strdup("Das"); name.str = (char *) malloc(strlen(fName.str) + strlen(lName.str) + 1); strncpy(name.str, fName.str); strcat(name.str, lName.str); cout << "First Name: " << fName.str << endl; cout << "Last Name: " << lName.str << endl; cout << "Full Name: " << name.str << endl; return 0; }</pre> <p>First Name: Partha Last Name: Das Full Name: Partha Das</p>	<pre>#include <iostream> #include <cstring> using namespace std; typedef struct _String { char *str; } String; String operator+(const String& s1, const String& s2) { String s; s.str = (char *) malloc(strlen(s1.str) + strlen(s2.str) + 1); strcpy(s.str, s1.str); strcat(s.str, s2.str); return s; } int main() { String fName, lName, name; fName.str = strdup("Partha "); lName.str = strdup("Das"); name = fName + lName; // Overload operator + cout << "First Name: " << fName.str << endl; cout << "Last Name: " << lName.str << endl; cout << "Full Name: " << name.str << endl; return 0; }</pre> <p>FIRST Name: Partha Last Name: Das Full Name: Partha Das Partha Pratim Das</p>

NFTEL MOOCs Programming in C++

So, exciting part is C++ now allows us to allow the user to define an operator function as I have shown you and to overload it; that is, I can write my own algorithm for this

operator function based on MyType. Let me take an example, the example looks little long do not worry this is a simple example. The objective here is to show that, really if I use the operator overloading how am I expected to benefit from that. So, this example shows the code for concatenating two strings, you know concatenation of strings, if we have two strings we want to put the second string after the first string put them together, we say this is a concatenation. So, on the left, if you look into the left column, this is what we assume is the type of the string, this is just something some type I am defining which is nothing, but a pointer to character. So obviously, the string will actually have to be dynamically allocated.

Now, my objective is to take two names of a person fName standing for the first name lName standing for the last name and to compose the total name by concatenating the lName after fName. So, just to keep things simple I show a hard-coded example. So, fName.str, which is the string component in the structure that have defined for the type, I duplicate a constant string by a strdup. Similarly, for lName.str in that, other string I duplicate another, the last name part of the intended name. Now, if I have to put the lName after the fName, then what I need to do, I need to concatenate these two strings in terms of a new string. So, the first thing that I will require is a new string name, which is went with the concatenation must have enough space allocated.

So, this is the typical code, you will do to allocate, you will find out the length of the fName, find out the length of the lName add one to that length, because you need the extra space for the null and malloc. I did not multiply this by sizeof, because I am assuming the size of the character is one. So, malloc and cast to char*. So, you have enough space in the name and then you do this these two typical steps, that is you copy the first name to name.str and then copy lName.str; then concatenate, I am sorry, not copy, concatenate strcat, the lName.str to the name.str. So, name.str already had the fName; and at the end of it, the lName.str will get concatenated. And if you output, you will get the concatenation happening which is here. So, this is the traditional way of doing it.

Now, look at how we want to do this in C++ using the operator function or the overloading of the operator function. We say that conceptually concatenation can be

thought of as an addition of two strings; that is, it is putting one string after the another and therefore, if I can take two strings and be able to write it as string one plus string two is my result, then I it would be minimize expression wise it will very convenient. So, look at the main function in the right-hand side column, we have the same set of variables, we have the same initialization here that is fName.str and lName.str are duped from constant strings. But then to concatenate we just express that, we should be able to write fName plus lName and that result should be the concatenated name together.

Just to show you to do this whole thing here, I need to write this code which is; this part of the code is, what is, addition. Certainly, this on the left-hand side, the code, I had to write is not only is longer, but has to be written carefully taking care of the strlen. And remembering that the first string has to do strcpy why the second has to do strcat here just conceptually you just add. Now, how does it work, to make this work, what I need to do is; I need to define an overload of operator plus for the string type that have defined. So, I define it on the top it takes two strings, certainly I do not want these to be tempered in anyway. So, I pass them as constant reference and within the function within this operator function that have implemented I allocate enough memory I copy the first string, concatenate the second; this basically is the code that I was writing here actually goes here.

But the advantage is now this code has been written only once. Now no matter how many times I want to concatenate strings, I do not need to write this code over and over again, where as in case of the style shown in the left-hand side, if I have now two other strings different strings, I need to do the tried this code again. So, that increases the error, that reduces the readability of this. Here after this has been done if I write it with operator notation of plus then, what it means is the compiler finds out that for the operator plus is there any operator function which takes on the left and right two string type variables, because fName is of string type lName is of string type. It finds do I have an operator plus, where the left-hand side is string type fName, right-hand side is also string type which is lName and if it exists then it will simply call this function passing fName as s1 and lName as s2.

Then this function computes the value and returns the result which is copied to the name. So, this gives us the huge advantage in terms of being able to write very compact algebraic kind of expressions by defining our own operators for our own type and the length that can go is huge. For example, we know I mean we often do matrix algebra, we add matrices, we multiply matrices, we invert matrices, and we take difference of matrices and so on. Now there is no matrix type, it is just an array, it is a two-dimensional array. So, if I have to do it in C I have to write all different functions and a simple addition of two matrices will turn out to be a major function call and all those details will come in. Whereas, if I could overload in the way we have shown operators like plus for adding two matrices, operator like binary multiplication * (star) to multiply two matrices and so on, and the whole life of the programmer will get much easier, so that is a basic objective.

(Refer Slide Time: 23:22)

w/o Overloading +	Overloading operator +
<pre>#include <iostream> using namespace std; enum E {C0 = 0, C1 = 1, C2 = 2}; int main() { E a = C1, b = C2; int x = -1; x = a + b; cout << x << endl; } return 0; </pre> <ul style="list-style-type: none"> Implicitly converts enum E values to int Adds by operator+ of int Result is outside enum E range 	<pre>#include <iostream> using namespace std; enum E {C0 = 0, C1 = 1, C2 = 2}; E operator+(const E& a, const E& b) { unsigned int ua = a, ub = b; unsigned int x = (ua + ub) % 3; return (E)x; } int main() { E a = C1, b = C2; int x = -1; x = a + b; cout << x << endl; } return 0; </pre> <ul style="list-style-type: none"> operator + is overloaded for enum E Result is a valid enum E value

I will just for interest show another simple interesting example, where I show how certainly you can change the semantics of these operators in certain context. On the left if you focus, the basic type that I am going to use here is an enum type e, if that is an enumeration which basically the sub-range of selective sub-range of integer type. Where I define three enumerated constants C0, C1 and C2 with the constant value 0, 1 and 2. So, in this enum world there are only three symbols. So, it is a domain of three symbols

three values, then here I define two variables a and b with c1 and c2, that is they will actually have values 1 and 2 and then I add a with b and put that value to x.

Now what will happen, if I add a and b enum as you know is the sub range of int, so if I try to add one enum with another naturally there is no separate addition of operation for enum. So, what it does it simply implicitly converts the enum to int and adds by the operator of int. So, what will happen this value is c1, a is c1 which is 1, b is c2 which is 2. So, you add these two the result is 1 plus 2, 3 and you assign that to x and you see that you have output a value 3.

Now, while according to the C program this is a correct thing do, but we feel somewhat uncomfortable because we wanted to remain in the domain of the enum that we have defined. By computing an addition of C1 and C2 possibly what we wanted to do is to wrap around because you go beyond the three constant 0, 1 and 2 that you had and we probably would have like to wrap around so that if you are adding 1 to 2 it should actually become 0 that is it should on increment come here and keep on going like that, but because the addition has taken me to integer, I have a result 3, there is no corresponding enum constant for that in my type definition. So, I am not happy with this. So, I want to define my own addition, where if I add two enums from my domain that is C0, C1 and C2 this is again the same enum type, what I want is; the result will always be within this enum type.

So, naturally if you add 0 with 0, it is within, if you add 1 with 1, it is 2, it is within, 0 plus 1, 1 plus 0, all these are within. But if you add 1 with 2, it should become 0 that is it should wrap around and come to 0. Similarly, if you add 2 with 2, it should become 1; if I add 2 plus 2 then it should become 1, so that is a kind of definition that I am looking at. So, it is a wrap around that I want. So, this is this is all the same this code is exactly the same the main routine has not been changed. Only thing that I do in between is, I introduce a operator+ (operator plus) function for this enum type; which now takes two different enum parameters, which are basically these two parameters in the invocation of addition and returns an enum. And what do I do, I do a very simple trick instead of just adding, which is the behavior that we were getting here, instead of just adding, we add and then modulo 3, 3 is a number of symbols you have.

So, what will happen if you add 1 with 2 this addition is 3. So, if when you do modulo with 3 then the result will turn out to be 0 that is it will wrap around, which is a very common technique you know that the remainder function, your remainder operator, the modulo operator always gives you to get that wrap around. So, we just use the trick and then take again take that value and put it back as at the return statement to be given back here.

So, now if I do a plus b the compiler makes out that a is of type e, b is also of type e. So, it tries to find out an operator plus function which takes an e and then e and that is what it finds here. So, on this $a+b$ (a plus b) it no more converts a, and b to integer and performs an integer addition rather it invokes his function and performs this operation, which truly preserves the closeness of my enum domain and gives me a value, which is 0 which is what I wanted. So, the operator plus is thus overloaded in enum and allows me to restrict the value within my enum range.

So, you can see that we have just shown two examples of somewhat different kinds, where in both cases, the operator plus has been overloaded. In string it allows me to do a replace complex string functions by just using a plus operation to concatenate. And here for enum, I can give a special semantics, I can give a new semantics for my type, while the underline type, you may continuous to remain to be unbuilt in integer.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 16
Operator Overloading (Contd.)

Welcome to module 9 of Programming in C++. We have been discussing operator overloading. We have seen the similarity and differences between operators and functions and we have seen how in C++, features have been given to define operator functions and overload them, and with that we have taken two examples in the earlier part to overload operator plus (operator +) for a string type that we have defined and concatenate two strings and we have also overloaded operator plus in a different context for an enum type to show how we can have a closed add operation for enum types.

(Refer Slide Time: 01:04)

The slide has a blue header bar with the title "Operator Overloading – Summary of Rules". On the left, there is a vertical navigation menu with the following items: Module 09, Partha Pratim Das, Objectives & Outcome, Operator & Functions, Operator Overloading Examples, String Class, Operator Overloading Rules, and Summary. The main content area contains a bulleted list of rules:

- No new operator such as **, <>, or &| can be defined for overloading
- Intrinsic properties of the overloaded operator cannot be changed
 - Preserves arity
 - Preserves precedence
 - Preserves associativity
- These operators can be overloaded:
[] + - * / % & | ~ ! = += -= *= /= %= &= |=
<< >> <<= >>= != < > <= >= && || ++ -- , ->* -> () []
- For unary prefix operators, use: MyType& operator++(MyType& s1)
- For unary postfix operators, use: MyType operator++(MyType& s1, int)
- The operators :: (scope resolution), . (member access), .* (member access through pointer to member), sizeof, and ?: (ternary conditional) cannot be overloaded
- The overloads of operators &&, ||, and , (comma) lose their special properties: short-circuit evaluation and sequencing
- The overload of operator-> must either return a raw pointer or return an object (by reference or by value), for which operator-> is in turn overloaded

At the bottom of the slide, there is a footer bar with the text "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" on the left, and the number "9" on the right.

Now, we will move forward to take a more detailed look into what can be overloaded and how and what if you are defining your own type, any type then what are the operators you can overload and how you can do that overload? What is the possible? What is not possible? What is advisable and so on?

So, I present here a summary of rules. Let us go through them very carefully. The first naturally; if we have to overload your first question is - what is the operator that can overload? Certainly, a number of operators are given in C++; plus, minus, division, multiplication, assignments, various kinds of extensions of assignment, all of those exist. So, can you define a new operator symbol and overload that? If that is a question if you have then the answer is no; that is you have to restrict yourself only to the set of operators that are defined in the system.

For example, you cannot say that I have a overloading; for say this operator or I will have this operator. Actually, some of you had been the old programmers in Pascal, you will recognize that this was the inequality in Pascal, but this is not a defined symbol in C++. So, you cannot overload operators with new symbols.

The second point which is very important is, when you overload an operator; you cannot change its intrinsic properties. Intrinsic properties will have to remain same and there are 3 intrinsic properties for an operator; that is arity, there is a number of operand it takes, the precedence in with respect to other operators and associativity with respect to its own group of operators or operators of the same precedence. These 3 intrinsic properties cannot be changed.

(Refer Slide Time: 03:08)

Operator Overloading – Summary of Rules

- No new operator such as `**`, `<>`, or `&|` can be defined for overloading
- Intrinsic properties of the overloaded operator cannot be changed
 - Preserves arity
 - Preserves precedence
 - Preserves associativity
- These operators can be overloaded:

$$\begin{array}{l} a + b \\ a - b \\ a * b \\ a / b \\ a \% b \end{array}$$

$$\begin{array}{l} a == b \\ a > b \\ a < b \\ a >= b \\ a <= b \end{array}$$

$$\begin{array}{l} a \& b \\ a \&= b \\ a \&= b \end{array}$$

$$\begin{array}{l} a \oplus b \\ a \ominus b \\ a \otimes b \\ a \oslash b \end{array}$$

$$\begin{array}{l} a \& b \\ a \&= b \\ a \&= b \end{array}$$
- For unary prefix operators, use: `MyType& operator++(MyType& s1)`
- For unary postfix operators, use: `MyType operator++(MyType& s1, int)`
- The operators `::` (scope resolution), `.` (member access), `.*` (member access through pointer to member), `sizeof`, and `?:` (ternary conditional) cannot be overloaded
- The overloads of operators `&&`, `||`, and `,` (comma) lose their special properties: short-circuit evaluation and sequencing
- The overload of operator `->` must either return a raw pointer or return an object (by reference or by value), for which operator `->` is in turn overloaded

So, if you argue that, I do have an operator plus, which can be written like this or it can be written like this. I have an operator minus which can be written like this, which can be written like this and so on. So, which means here it is, arity is 2, here the arity is 1, but just such kind of changes you will not be able to do yourself. If for the operator multiple versions of arity exist and correspondingly different precedence and associativity are defined, you will have to go by that, but you cannot; for any of the operators define or change its arity precedence and associativity. So, intrinsic properties will have to be totally honored.

The third are a list of; these are the commonly used 38 operators in C++ which can be overloaded. So, you can see that you have almost all the operators that you can think of, including the basic arithmetic operators and a whole lot of assignment operators, then your shift operators, your logical operators, your pointer referencing operator, your array operator, your function operator and so on. So, all this can be overloaded.

(Refer Slide Time: 04:43)

Operator Overloading – Summary of Rules

- No new operator such as `**`, `<>`, or `&|` can be defined for overloading
- Intrinsic properties of the overloaded operator cannot be changed
 - Preserves arity
 - Preserves precedence
 - Preserves associativity
- These operators can be overloaded:
 - `[] + - * / % & | ^ ! = += -= *= /= %= |= |=`
 - `<< >> >>= <<= != < > <= >= && || ++ -- , ->> -> () []`
- For unary prefix operators, use: `MyType& operator++(MyType& s1)`
- For unary postfix operators, use: `MyType operator++(MyType& s1, int)`
- The operators `::` (scope resolution), `.` (member access), `.*` (member access through pointer to member), `sizeof`, and `?:` (ternary conditional) cannot be overloaded
- The overloads of operators `&&`, `||`, and `,` (comma) lose their special properties: short-circuit evaluation and sequencing
- The overload of operator `->` must either return a raw pointer or return an object (by reference or by value), for which operator `->` is in turn overloaded

If you have a unary operator, as you know unary operators are of two types, one are prefix operators which happen before the operand, that is this kind of; these are all prefix operator or they can be of the post fix type.

(Refer Slide Time: 04:47)

Page 28 / 28

Operator Overloading – Summary of Rules

Module 09
Partha Pratim Das

Objectives & Outline
Operators & Functions
Operator Overloading
Examples
String Class
Operator Overloading Rules

W

- No new operator such as **, <>, or &| can be defined for overloading
- Intrinsic properties of the overloaded operator cannot be changed
 - Preserves arity
 - Preserves precedence
 - Preserves associativity
- These operators can be overloaded:
[] + - * / % & | ~ += -= *= /= %= |=
<< >> <<= |= < > <> && || ++ -- , ->> -> () []
- For unary prefix operators, use: `MyType& operator++(MyType& s1)`
- For unary postfix operators, use: `MyType operator++(MyType& s1, int)`
- The operators :: (scope resolution), . (member access), * (member access through pointer to member), sizeof, and ?: (ternary conditional) cannot be overloaded
- The overloads of operators &&, ||, and , (comma) lose their special properties: short-circuit evaluation and sequencing
- The overload of operator-> must either return a raw pointer or return an object (by reference or by value), for which operator-> is in turn overloaded

So, the question is a same operator particularly, if you look at ++, I can write ++a or I can write a++. The question naturally is given the correspondence between an operator and then operator function, we have said that the operator function corresponding to an operator is just the operator keyword. So, it is operator keyword followed by the operator symbol. So, both of these will necessarily have the same operator function name.

So, your question will be, but they are different operators, prefix and post fix are different operator, pre-increment and post-increment are different behaviors. So, how do we distinguish that? So, in these two points you will find that answer that if a unary operator is prefix, then like this you simply write it, the signature will look something like this, which takes one operand because it is unary of your type and it returns the operand that it had taken after the incrementation.

Whereas, if it is post fix operator then the interesting thing is you will have to specify a int as a second parameter and this int actually is not an active parameter. That is when you actually invoke the operator there is no int that you are going to pass it is just there in the signature. but this helps the compiler to decide that this operator this instance of the operator function is for the post fix type and not of the prefix kind. So, if I write a++

then it minds to this. If I write `++a`, it minds to this. This is the basic mechanism of resolving prefix and post unary operators for overloading.

Next, please note that there are some operators which are not allowed to be overloaded like scope resolution operator, like member access, for example, if we have a structure, we had seen complex structure with `re` and `im` as components. So, I can take the structure name put a dot and put `re`. So, that is the `re` component of the structure. So, the member access cannot be overloaded. The `sizeof`, to find the number of bytes of any variable or type cannot be overloaded; the ternary operator cannot be overloaded and so on.

There are some operators which are allowed to be overloaded like logical AND, logical OR, comma etc, but you have to keep in mind that if you overload them then their basic properties, some additional properties might get destroyed.

(Refer Slide Time: 07:53)

Module 09
Partha Pratim Das
Objectives & Outline
Operators & Functions
Operator Overloading
Examples
Assignment
Operator Overloading Rules
Summary

Operator Overloading – Summary of Rules

- No new operator such as `**`, `<>`, or `&|` can be defined for overloading
- Intrinsic properties of the overloaded operator cannot be changed
 - Preserves arity
 - Preserves precedence
 - Preserves associativity
- These operators can be overloaded:
`[] + - * / % & | ^ ! = += -= *= /= %= &= |=`
`<< >> >>= <<= != < > <= > && || ++ -- , ->* -> () []`
- For unary prefix operators, use: `MyType& operator++(MyType& s1)`
- For unary postfix operators, use: `MyType operator++(MyType& s1, int)`
- The operators `::` (scope resolution), `.` (member access), `*` (member access through pointer to member), `sizeof`, and `?:` (ternary conditional) cannot be overloaded
- The overloads of operators `&&`, `||`, and `,` (comma) lose their special properties: short-circuit evaluation and sequencing
- The overload of operator `->` must either return a raw pointer or return an object (by reference or by value), for which operator `->` is in turn overloaded

That is in ampersand i.e. 'logical and' if I write this expression `(a == b && b == c)`, then this logical and has a behavior that if this part of the expression gets false then it does not evaluate the second part. If this part of the expression gets false then it does not evaluate the second part, do you see the correctness of the logic. If `a` is not equal to `b`; `a` and `b` are different then `a` equal to `b` will become false and once this become false, since it is an

AND operation no matter what that truth or falsity of b being equal to c whether the second part is true or second part is false, this whole expression is going to be get false anyway.

So, this kind of just evaluating one part and not evaluating the other is known as the short circuit in evaluation, and is done by sequencing, because you need to decide in which order you evaluate them. These are special behaviors of this operator. So, if you overload those behaviors will get lost, you will have to be careful that after you overload those behaviors cannot be assumed.

Finally, if you overload the pointer indirection operator then that operator must return another pointer either it is a directly a pointer. This point just you note, it is not easy to understand at this stage. We will at some point talk about particularly overloading of this operator, which is a very strong feature of C++ programming not though of the language. It is called smart pointers and that use it. So, if you are overloading this operator then you need to necessarily return a pointer or something that can become again a pointer.

These are the basic rules of operator overloading. So, following this you can start writing your operators and I have shown examples as in string and enum that, you really thought the rules have to be strictly followed, but they are quite intuitive and straight forward and taking the examples that I have discussed here. You can simply write more and more types of your own, it would be good to write a full complex type where you could actually slowly overload other operators and really make the complex numbers behave as in the integers once.

(Refer Slide Time: 10:36)

The slide has a blue header with the title 'Overloading disallowed for'. On the left, there's a vertical sidebar with navigation links: Module 09, Partha Pratim Das, Objectives & Outline, Operators & Functions, Operator Overloading Examples, String Class, Operator Overloading Rules, and Summary. The main content area contains a table with two columns: 'Operator' and 'Reason'. The operators listed are dot (.), Scope Resolution (::), Ternary (? :), and sizeof. The reasons are: dot (.) raises a question about object reference or overloading; Scope Resolution (::) performs scope resolution rather than expression evaluation; Ternary (? :) would not guarantee execution of expr2 and expr3; and sizeof cannot be overloaded because built-in operations like incrementing a pointer depend on it.

Operator	Reason
• dot (.)	• It will raise question whether it is for object reference or overloading
• Scope Resolution (::)	• It performs a (compile time) scope resolution rather than an expression evaluation.
• Ternary (? :)	• overloading expr1 ? expr2 : expr3 would not be able to guarantee that only one of expr2 and expr3 was executed
• sizeof	• Sizeof cannot be overloaded because built-in operations, such as incrementing a pointer into an array implicitly depends on it

NPTEL MOOCs Programming in C++ Partha Pratim Das 10

In the following slides, which I will not go through in detail in the lectures, I have tried to put down the operators which are not allowed to be overloaded as I had mentioned and I have tried to provide a reasons so that you not only have; we just do not simply need to remember is to why it is not allowed to overload the ternary operator or why is it not allowed to overload the sizeof operator, you can actually understand the reason and then it will be easier for you to remember.

(Refer Slide Time: 11:12)

The slide has a blue header with the title 'Do not overload these operators'. On the left, there's a sidebar with navigation links for 'Module 09', 'Partha Pratim Das', 'Objectives & Outline', 'Operators & Functions', 'Operator Overloading', 'Examples', 'String Class', 'Operator Overloading Rules', and 'Summary'. The main content area contains a table with three columns: 'Operator', 'Reason', and a third column which is partially visible. The 'Operator' column lists '&& and ||', 'Comma (,)', and '&'. The 'Reason' column provides detailed explanations for each operator.

Operator	Reason
• && and	• In evaluation, the second operand is not evaluated if the result can be deduced solely by evaluating the first operand. However, this evaluation is not possible for overloaded versions of these operators
• Comma (,)	• This operator guarantees that the first operand is evaluated before the second operand. However, if the comma operator is overloaded, its operand evaluation depends on C++'s function parameter mechanism, which does not guarantee the order of evaluation
• Ampersand (&)	• The address of an object of incomplete type can be taken, but if the complete type of that object is a class type that declares operator &() as a member function, then the behavior is undefined

NPTEL MOOCs Programming in C++ Partha Pratim Das 11

So, this one list which is overloading, where overloading is disallowed and another list which I briefly discussed is where there are operators, where overloading is actually allowed, but it is advised that you do not overload them because if you overload them then some of the very basic behaviors of C++ program changes. So, these are to be overloaded by really expert people and till you get to that level, I would advice that you please do not overload that.

The remaining 38 operators that you have at hand, is rich enough to develop any kind of algebra, with any kind of matrix, complex, fraction any kind of types that you want to.

(Refer Slide Time: 11:56)

The slide has a dark blue header bar with the title "Module Summary". On the left, there is a vertical sidebar with a logo at the top and a list of navigation links: "Module 09", "Partha Pratim Das", "Objectives & Outline", "Operators & Functions", "Operator Overloading", "Examples", "String Class", "Operator Overloading Rules", and "Summary". The main content area contains a bulleted list: "• Introduced operator overloading" and "• Explained the rules of operator overloading". At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" along with the number "12".

With this we come to the close of this module where, here we have introduced the operator overloading and we have explained different rules and exceptions for operator overloading.

And in the next module, in module 10, we will show another special case of extension and special case of operator overloading in terms of dynamic memory management. We will introduce the operators that C++ provide for dynamic memory management and we will again show that how operator overloading can be applied in the context of dynamic memory management operators to get, see various kinds of strong advantages in memory management in C++.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 17
Dynamic Memory Management

Welcome to module 10 of programming in C++. We have been discussing the procedural extensions of C for C++, this is the part of better C of C++, and we have already discussed several features starting from const, inline function, reference parameter and so on. And this will be the last in this series. Today, we will discuss about dynamic memory management in C++.

(Refer Slide Time: 01:05)

The slide has a blue header bar with the title "Module Objectives". On the left, there is a sidebar with a logo and navigation links for "Module 10" and "Objectives & Outline". The main content area contains a bullet point: "• Understand the dynamic memory management in C++". At the bottom, it shows "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" along with a page number "2".

So, we are trying to understand the dynamic memory management capabilities of C++ over what already exist in C.

(Refer Slide Time: 01:16)

This slide shows the module outline for NPTEL MOOCs Programming in C++. The title is "Module Outline". On the left, there's a sidebar with "Module 10" and "Partha Pratim Das" at the top, followed by a list of topics: "Objectives & Outline", "Memory Management in C", "malloc & free", "Memory Management in C++", "new & delete", "Array new[] and delete[]", "Placement new()", "Restrictions", and "Overloading new and delete". Below the sidebar is a circular profile picture of Partha Pratim Das. The main content area contains a bulleted list of topics under "Memory management in C++": "new and delete", "Array new[] and delete[]", "Placement new()", and "Restrictions". At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

So, this will be the outline, as usual, you can see the outline on the left border as well. The memory management in C, we will recap with malloc and free, and then we will talk about various ways that the management can be done in C++, and how is that better than what we can do in C.

(Refer Slide Time: 01:40)

This slide compares C and C++ programs for dynamic memory management. The title is "Program 10.01/02: malloc() & free(): C & C++". It shows two side-by-side code snippets. The C program includes `#include <stdio.h>` and `#include <stdlib.h>`. The C++ program includes `#include <iostream>` and `#include <cstdlib>` with `using namespace std;`. Both programs define a `main()` function that allocates memory for an integer, prints its value, and then frees it. Below the code, a bulleted list explains the differences: "Dynamic memory management functions in stdlib.h header for C (cstdlib header for C++)", "malloc() allocates the memory on heap", "sizeof(int) needs to be provided", "Pointer to allocated memory returned as void * - needs cast to int *", "Allocated memory is released by free() from heap", and "calloc() and realloc() also available in both languages". The slide also features a sidebar with "Module 10" and "Partha Pratim Das" at the top, followed by a list of topics: "Objectives & Outline", "Memory Management in C", "malloc & free", "Memory Management in C++", "new & delete", "Array new[] and delete[]", "Placement new()", "Restrictions", and "Overloading new and delete". A circular profile picture of Partha Pratim Das is also present. The bottom of the slide says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

So, we will start with an example on the left column, you can see a very simple program which is dynamically allocating a memory. So, if we look at, this is the place where the allocation is happening. So, p is a pointer to an integer and we are using the malloc function which will allocate the memory on heap and the malloc function is available in stdlib.h. So, you have included that header. And as you are aware malloc needs the size of the memory to be allocated, which is basically the number of bytes that I need for storing the data that I want to keep.

Since, I want to keep an integer, we store the size of we pass the size of int to malloc this is what we are doing here. Malloc in return; so this is this is a point we have highlighted it below also. And with this parameter, malloc will dynamically allocate the appropriate number of bytes as a consecutive chunk, they all will be together as a consecutive chunk on the heap. So, this is an important point that here when we usually talk of dynamic memory allocation, we are talking of allocating the memory in certain segment of the memory, typically known as heap or some places this is called as a free store, this is the memory that we can use freely for your purpose.

Now, once a call to malloc is done then malloc returns a void* pointer, malloc will return a void star pointer that because malloc does not know what type of data you are going to store in the memory that malloc is allocating for you. So, malloc will return you a point at to an unknown type of data by a void*. And it is a responsibility of the programmer as in here, where we will need to cast that pointer with the appropriate type, since we want an integer to be stored look at the left-hand side, we will cast this as int *. This will give us, so with this, what will have is we will have something like this; this is my p and this is the memory that has been allocated. So, this is a pointer this memory has been allocated.

So, if the address where this memory has been allocated is 200, then p will have a value 200. And then I want to dereference, that is I want to traverse the point at p and store a value at this integer location. So, as I do *p assign 5, 5 will go in to this memory location, which can subsequently be used in the print statement, and this will print the value 5. So, we all understand that and finally, when I am done when we do not need that dynamically allocated memory anymore, we will free it up by a call to the free function.

This will be released by a call to the free function. And this will be released from the heap from where this was originally allocated. So, this is the mechanism of dynamic allocation for a variable using malloc and free functions available in the standard library.

Now, certainly due to backward compatibility, C++ programs will also be able to use it. So, as we have already explained several times. Now the header name will become cstdlib, this will give everything in the std namespace. So, we are saying, using namespace std that is we will every standard library symbol is prefixed with a std:: as if. And we write exactly the same code as in the left-hand side. If you compare them side-by-side they are exactly the same code, and this code will run in C++ exactly as does in C.

So, what we understand is the dynamic allocation and deallocation or management of memory, using malloc and free can be done exactly in the same way in C++ as well. Here I have been discussing about malloc for allocation alone, you know there are other allocation functions like calloc, and realloc; realloc is often used for extending a given memory and so on. So, these functions can also be equally used in C++, but we will not show explicit example of them, we will just restrict to malloc.

(Refer Slide Time: 06:42)

**Program 10.02/03: operator new & delete:
Dynamic memory management in C++**

Module 10
Partha Pratim Das

Objectives & Outline
Memory Management in C
malloc & free
Memory Management in C++
new & delete
free
Placement new
Overloading new & delete
Summary

C++ introduces operators new and delete to dynamically allocate and de-allocate memory:	
<pre>#include <iostream> #include <cstdlib> using namespace std; int main() { int *p = (int *)malloc(sizeof(int)); *p = 5; cout << *p; free(p); return 0; }</pre>	<pre>#include <iostream> using namespace std; int main() { int *p = new int(5); cout << *p; delete p; return 0; }</pre>
<ul style="list-style-type: none"> Function malloc() for allocation on heap sizeof(int) needs to be provided Allocated memory returned as void * Casting to int * needed Cannot be initialized Function free() for de-allocation from heap Library feature – header cstdlib needed 	
<ul style="list-style-type: none"> Operator new for allocation on heap No size specification needed, type suffices Allocated memory returned as int * No casting needed Can be initialized Operator delete for de-allocation from heap Core language feature – no header needed 	

NPTEL MOOCs Programming in C++ Partha Pratim Das 5

Now, let us take a look at, what C++ offers besides malloc and free that already comes to C++ from C. So, the program on the left-hand side is the program that we have just seen, the C program using malloc and free. The program on the right-hand side is using C specific mechanism of C++. Focus on this line, where a memory is being allocated. C++ introduces a new operator; the name of that operator is operator new or new simply. So, that if you write new and write a data type rust after this operator new then new will do the same task that malloc was doing that is new will also request the memory manager of the system to give it some free memory from the heap and return you the pointer to that memory. So, the basic functionality of new is exactly like malloc. But there are several distinctions, several differences that exist between the way new needs to be used and the way malloc is traditionally used or the malloc can be used.

First thing is malloc needs the size of the memory that we want to be passed as a parameter to malloc, new does not need anything like that. New in contrast, actually takes the type of the variable that I want to create the space for, the type to be passed as a parameter. Passing a type as a parameter is a relatively difficult concept for you at this stage, because you have never seen a type being passed. So, for now, just take it by heart that new is an operator, which instead of taking a value, it can actually take a type. So, here new operator is taking the type int. And since it takes the type int, it can internally compute, how much space it requires, it can internally do the sizeof on this int that malloc needs to be explicitly passed, and therefore, it does not need any size specification to be given. So, in malloc, you need to give the size in new, you do not need to pass any size.

The second major distinction is that, the malloc since it does not know what type of data you are going to keep in the memory that it is allocating, it cannot give you a pointer of an appropriate type. It gives you a pointer which is void* pointed to some unknown type, which is we have already discussed you know. But in new, as I have just mentioned new takes the int type as a parameter. So, new knows that, you are going to keep an int type value in the memory that new is allocating therefore, new will return you a pointer of the int* type, the actual type that you need. So, that is a something very-very interesting because here the example is showing with new if you now do a new with double it will return you a double star pointer. If you do new with say a structure type complex, then it

will return you a complex star kind of pointer. So, new will knows what type of value you are going to keep and therefore, it returns to a pointer of that appropriate type. So, note on this point that, since malloc returns you a void *, you will need to cast it to int *; new in contrast will directly give you an int * pointer and no casting is required.

Next after the allocation was done, in this case, we went ahead and put an initial value in the location that has been allocated, we had assigned 5 to *p. Now using malloc, there is no way to allocate and initialize together. New gives you even that facility that is if I want I can pass this initial value five as I am doing here, so that when new returns me the pointer, it will not only give me the pointer of the location allocated, but that location will already carry the initial value five that I want there. So, if we just see the dynamics, if I just draw them again this is p on this side of the program and this is the allocation. So, once this is done, you will get a state like this where you do not know what value exist. And then you will do *p assigned 5 which will take 5 and put it here. So, there is no initialization in this case, you are making an assignment afterwards through the second statement.

Whereas, here when this will get executed, when you get back the pointer p, you get back the whole thing in this situation, that the pointer p is pointing to their allocated location. And the value already is 5. So, that is a very good advantage; that in malloc initialization is not possible, in new the initialization can be done. Of course, you have the choice of, if you would like to just allocate and not initialize, you can even do all that. you will need to do is; write it as new int and not give the initial value; in that case it will give you an uninitialized value at malloc.

Finally, when you are done, if you had allocated the memory by malloc, you need to free it release it by a call to the free function. Here, you have another operator which is also new for C++, another operator the name of this operator is delete. So, you write the operator delete and then put the pointer. So, this will release the memory back to the system as free does. They are very similar in terms of. So, free was deallocating in the memory from the heap, and operator delete will similarly deallocate the memory from heap.

In all this discussion of a new and delete, please remember that I am consistently saying that malloc and free are functions, whereas, new and delete are operators. So, while we discussed about operator overloading, we gave you several distinctions between an operator and a function. So, those distinctions will exist between operator new and function malloc, or operator delete and function free and we will make use of some of those distinctions as we go forward. Finally, you can also note that malloc and free are not part of the core language. So, they come from the standard library cstdlib, whereas, new and delete are part of the core C++ language therefore, to do new or delete you do not need any special header to be included.

(Refer Slide Time: 14:53)

**Program 10.02/04: Functions:
operator new() & operator delete()**

Module 10 Partha Pratim Das	
<ul style="list-style-type: none"> Objectives & Outline Memory Management in C Memory Management in C++ new & delete Arith. Operators and Expressions Overloading new & delete Summary 	<ul style="list-style-type: none"> C++ also allows operator new and operator delete functions to dynamically allocate and de-allocate memory:
malloc() & free() <pre>#include <iostream> #include <cstdlib> using namespace std; int main() { int *p = (int *)malloc(sizeof(int)); *p = 5; cout << *p; free(p); return 0; } </pre>	new & delete <pre>#include <iostream> #include <cstdlib> using namespace std; int main() { int *p = (int *)operator new(sizeof(int)); *p = 5; cout << *p; operator delete(p); return 0; }</pre>
<ul style="list-style-type: none"> Function malloc() for allocation on heap Function free() for de-allocation from heap 	<ul style="list-style-type: none"> Function operator new() for allocation on heap Function operator delete() for de-allocation from heap

There is a major difference between operator new and function operator new(). We explore this angle more after we learn about classes

NPTEL MOOCs Programming in C++ Partha Pratim Das 6

Now there is something very, very interesting. As we have the operator new and delete, similarly we can also write the operator functions for new and delete. We know that every operator has a corresponding operator function. So, it is possible that you use the operator function directly also. So, in this example, again on the left-hand side, we have the same malloc example with the free. On the right-hand side, instead of operator new, what we are showing is the use of the operator function new, the corresponding function, if you want to me to recall from the previous slide then when you use operator new you just write new and then pass the type. Whereas, when you use the operator function corresponding to new that is operator new function, it is pretty much like malloc. So, you

need to pass the size, you are no more passing the type, you need to pass the size, you need to cast the pointer back.

So, in C++, it is possible that you can also write malloc kind of call using the operator new function. And correspondingly you could also write in place of operator delete, which we had invoked earlier this is how you invoke operator delete, instead you could also invoke it in the style of free function. You can say operator delete which is a corresponding operator function and pass p as a parameter to the operator delete function. So, as malloc allocates on heap, operator new function will also allocate on heap, as free deallocate from heap operator delete function will also deallocate from heap. But please keep in mind that operator new and the operator new function are two very distinct entities, and there are major differences between operator new and the function operator new.

So, at the example up to which we have discussed no difference can be seen it appears as if I can either use operator new or I can use the function for operator new. If they are equivalent, they are just giving us the same effect. This is true as long as we use the built-in types. But, the moment we use some of the user define types we will see that the way operator new, the operator version works, and the way the corresponding function version works become very different. We will discuss them when we come to that stage, but we will explore that more, but right now we just I just want you to note that there is a major difference between them.

(Refer Slide Time: 18:12)

The slide title is "Program 10.05/06: Operators new[] & delete[]: Dynamically managed Arrays in C++". It features a logo of a person in a graduation cap on the left and a portrait of Partha Pratim Das in the center.

Module 10
Partha Pratim Das
Objective & Outline
Memory Management in C
malloc & free
Memory Management in C++
use & delete
Array
Pointers and
Dynamically
allocated
arrays

malloc() & free()

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main() {
    int *a = (int *)malloc(sizeof(int)* 3);
    a[0] = 10; a[1] = 20; a[2] = 30;

    for (int i = 0; i < 3; ++i)
        cout << "a[" << i << "] = "
            << a[i] << endl;
    cout << endl;

    free(a);

    return 0;
}
-----  
a[0] = 10    a[1] = 20    a[2] = 30
```

- Allocation by malloc() on heap
- # of elements implicit in size passed to malloc()
- Release by free() from heap

new[] & delete[]

```
#include <iostream>
using namespace std;

int main() {
    int *a = new int[3];
    a[0] = 10; a[1] = 20; a[2] = 30;

    for (int i = 0; i < 3; ++i)
        cout << "a[" << i << "] = "
            << a[i] << endl;
    cout << endl;

    delete [] a;

    return 0;
}
-----  
a[0] = 10    a[1] = 20    a[2] = 30
```

- Allocation by operator new[] (different from operator new) on heap
- # of elements explicitly passed to operator new[]
- Release by operator delete[] (different from operator delete) from heap

NPTEL MOOCs Programming in C++ Partha Pratim Das ?

Now, let us move on and look at what we dynamically allocate very often in C, we often allocate arrays. So, if we want to allocate arrays, we use the same malloc function, you all are familiar. So, this is the same malloc function. The only difference is we saying its size of int is multiplied by 3, which any experienced C programmer will understand that what we are trying to say is I want three consecutive areas of size int which basically mean that here this pointer should actually, eventually give me an array of 3 int elements.

This is how you read it, but if you look into what value malloc gets, malloc necessarily does not get to know what is size of int, or what is the value three malloc gets their product. So, if size of int is 4, the malloc will get a value 12. The malloc has no way to know that whether these are for 3 integers or it is for 12 unsigned characters and so on. It just gets a total cumulative value.

In contrast, operator new, and this has a different name, we call it array operator new. This array operator new will like before take the type which is a type of the element which is int, but then within a square bracket, it takes the number of elements that are required. So, it will know how many elements of int type to be created in the memory that needs to be allocated. So, behavior wise, it is very similar to malloc, but specification wise, it is making it very clear that; you can even see the uniformity of the

syntax you actually wanted to do this; int array of three integers (int[3]). And you literally write the same thing and as before it will return you not a void* pointer which needs to be cast in case of malloc, but it will return you an int* pointer.

Similarly, when we are done, you do free in case of a memory allocated by malloc, but in case of a memory allocated by array new, where you have used this number of elements, you pass the number of elements after that type, you will need to delete it by array delete (delete [] a;). The difference with the previous form is specifically need to put the array operator after the delete keyword and before the pointer. And then the array delete operator will release the memory to the heap. And please note that operator new and operator array new are different operators. Similarly, operator delete and the operator array delete are different operators; they can be handled extremely differently and we will show some examples of that.

So, this is the second type. So, it is advised that if you are dealing with individual data items then you just use operator new and operator delete, but if you are using arrays of data items then you use operator array new and operator array delete. So, that system explicitly know when we are dealing with an array, and when you are dealing with the single data item. This distinction was not possible in C using the standard library malloc free function, but here it is semantically gets absolutely clear.

(Refer Slide Time: 22:14)

The slide title is "Program 10.07: Operator new(): Placement new in C++". The left sidebar lists course objectives and a navigation menu. The main content shows the following C++ code:

```
#include <iostream> using namespace std;
int main() {
    unsigned char buf[sizeof(int)* 2]; // Buffer on stack ✓
    // placement new in buffer buf
    int *pInt = new (buf) int (3); int *qInt = new (buf+sizeof(int)) int (6);
    int *pBuf = (int *) (buf + 0); int *qBuf = (int *) (buf + sizeof(int));
    cout << "Buf Addr Int Addr" << endl;
    cout << pBuf << " " << endl << qBuf << " " << qInt << endl;
    cout << "1st Int 2nd Int" << endl;
    cout << *pBuf << " " << *qBuf << endl;
    int *rInt = new int(7); // heap allocation
    cout << "Heap Addr 3rd Int" << endl;
    cout << rInt << " " << *rInt << endl;
    delete rInt; // delete integer from heap
    // No delete for placement new
    return 0;
}
~~~~~
```

Below the code is a memory dump table:

Buf Addr	Int Addr
001BFC50	001BFC50
001BFC54	001BFC54
1st Int	2nd Int
3	6
Heap Addr	3rd Int
00200000	7

Hand-drawn annotations on the right side of the slide show a buffer structure with three integers. The first integer at address 001BFC50 is labeled '1st' and has value 3. The second integer at address 001BFC54 is labeled '2nd' and has value 6. A third integer at address 00200000 is labeled '3rd' and has value 7. Arrows point from the variable names in the code to their corresponding memory locations.

Now, we talk about yet another form of memory allocation in C++. This is known as placement new. This is another form of new operator, which allocates memory, but it has something which is very different in fundamentally in the concept. When we deal with malloc or for that matter operator new or operator array new, we want to allocate the memory and that memory we want has to come from heap, that is that memory has to be negotiated with the memory manager of the system and should come from the free store area free store segment of the memory. So, in that way, if we allocate any variable or an array in that manner, then certainly I do not have a control in terms of in which address or in which location this variable gets created.

But I may want to, and I will slowly explain the reason why I want to do this. I may want to do the same allocation, but I may want to do them in a memory address which is known to me, which I pass to this particular allocator. The situation is; let us suppose that I have a buffer. a buffer is nothing but you can you can just read the specification, you can read the definition of the buffer. It says the name is buff it is of type unsigned character and the size is or the number of elements of this buffer are sizeof(int) * 2, which means that if I am on a 32-bit machine then very typically sizeof(int) is 4. So, it is basically eight characters, eight bytes consecutive that I have as a buffer.

Now, what you want, I want to dynamically create an integer, but I do not want to create that integer in the free store. I want to create the integer in the buffer itself. So, if I try to drop the buffer then say, this is the buffer. So, these are the 8 bytes of the buffer this is buff, and since integer is of 4 bytes, so it is possible that I can take these 4 bytes and these 4 bytes to keep one integer here and another integer here and so on. That is these four consecutive bytes starting from, let say address, if this address is 200 then this address is 203, similarly this address will be 204, so this address goes up to 207, from 200 to 203 in that buffer I want to keep an integer; and the other one, I would keep.

So, I want to still dynamically allocate, I want to dynamically allocate, look at this line, I want to dynamically allocate an integer and get that pointer as pInt. But I have between the new and the int type, I have put the name of this buffer which tells me that please do not allocate it in a free store, allocate it from this buffer, that is buff is basically a pointer. So, value of buff is 200. So, what will this do, pInt, will allocate it the address from this buffer address of 200. And it will think of this part of the buffer as its integer. Similarly, if I next do qInt as in here, I am again allocating an integer, but I do not do it at buff, I do it at buff+sizeof(int). So, buff was 200, sizeof(int) is 4. So, buff+sizeof(int) is 204. So, if I do that allocation then qInt gets a value 204, that is it points here, and therefore, to qInt it is as if this is the other integer.

So, it is doing the allocations exactly in the same way, but with the difference that the allocations are not happening from the free store, it is happening from a buffer that already exist. Now, this buffer itself here I have taken the buffer to be of an automatic type, which is local to the function body, this buffer itself could be dynamically allocated in the free store or otherwise that we are not concerned with that. But that the reason we call this is a placement new, because it is not only doing an allocation, but it is also doing an allocation and placing it where I want it to be. So, this placement is the fact that I am not only doing the allocation, but I am passing a pointer saying that this is where memory is available you go and allocate it there, you go and place the object there.

Now, you will certainly argue that this is this could have been possible in C++ by writing various kind of pointer manipulations, why did I need to really do this, you will understand that, when we talk about the dynamic allocation of user define types. For

now, you just have to note that there are possibilities of doing a placement new. The only difference in case placement new from the other two forms of new is that, for placement new you have already provided that buffered. The memory was not dynamically allocated. Since, it was not dynamically allocated from the free store there is no sense of doing a delete there, because the memory manager did not actually get you that memory.

So, for this kind of a new, the placement new here or for the placement new here, you do not have a corresponding delete; otherwise, you can go through the rest of the example starting from initialization right here or the use in terms of other pointers. If you go through this code this is what a little bit of pointer tweaking, and it will be good to get accustom to that you will understand that rest of it is exactly like any other pointer manipulation. But the only difference is that the addresses are not coming from the dynamic store, not coming from the free store, addresses are coming from the buffer that I have provided.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 18
Dynamic Memory Management (Contd.)

Welcome to part-2 of module 10. We are discussing Dynamic Memory Management in C++. We have done a quick review of Memory Management Dynamic Memory Management in C, using Malloc and Free.

And then we have introduced 3 pairs of operators for Dynamic Memory Management in C++, the operator new and operator delete. The operator new can take a type and can also take an initial value and dynamically allocate a memory in free store, initialize it with the value given and then return a pointer of the type of value that has been created.

If you want an array of such values to be dynamically allocated in free store we have an operator new, which is of the array kind we say, array operator new and it has a matching array delete operator. And we have finally also shown that it is possible that we can do an operator new with a given address of memory that it is not necessary that the memory comes from the allocation by a memory manager, I can have a buffer that I already have either the buffer could be in the automatic area the stack area, the buffer could be in the global area, but I can pass the address of that buffer and create objects within that buffer.

Now, we will go ahead and discuss some of the restrictions that must be followed while you are doing this.

(Refer Slide Time: 01:52)

The slide has a blue header with the title 'Mixing malloc, operator new, etc'. On the left, there's a sidebar with a logo and a navigation menu for 'Module 10' by Partha Pratim Das, covering topics like Objectives & Outline, Memory Management in C, and Memory Management in C++.

The main content includes a bullet point about matching allocation and deallocation methods to avoid memory corruption, followed by a table comparing allocators and deallocator functions:

Allocator	De-allocator
malloc()	free()
operator new	operator delete
operator new[]	operator delete[]
operator new()	No delete

Below the table, there's a list of bullet points summarizing best practices for memory management:

- Allocation and De-Allocation must correctly match. Do not free the space created by new using free(). And do not use delete if memory is allocated through malloc(). These may result in memory corruption
- Passing NULL pointer to delete operator is secure
- Prefer to use only new and delete in a C++ program
- The new operator allocates exact amount of memory from Heap
- new returns the given pointer type – no need to typecast
- new, new[] and delete, delete[] have separate semantics

At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

So, since we have multiple different operators and multiple different operator as well as different functions to Allocate and De-allocate memory dynamically; we will have to be careful to note that the Allocation and De-allocation of any memory will always have to be done by the Allocator, De-allocator pair.

So, in the table I have listed these pairs and you will have to be very careful to follow that. If a memory location is dynamically allocated by malloc it must be released by free. If it has been allocated by operator new it must be released by operator delete. If it has been allocated by operator array new it must be released by the operator array delete and if it has been done true placement new then there is no need for doing a delete.

Now, any kind of other mixtures will lead to ambiguity and uncertain results. For example, if you allocate a memory by malloc and try to delete it by operator delete or allocate a memory by operator new and release it by free or you allocate a memory by operator array new and try to release it by simple delete and so on, any kinds of mixture combinations of Allocation and De-allocation that you may do, will lead to unpredictable disastrous results. So, be very-very careful to always use them in pairs.

And while you are Programming in C++, my advice is that, you always try to only use new and delete operators. It is better not to use malloc and free unless in very exceptional cases where their use is absolutely necessary. But, in a normal program you will not need

malloc and free to be used you can always do it operator new or operator array new and the corresponding delete function.

So, always delete operators, always try to do that and it is also possible that the delete operator can be passed in NULL pointer. So, you do not need to check, if you are trying to release a memory then that memory actually exists. If that memory has been NULL, the pointer has been Null even then the delete will work safely. So, while you do this Memory Allocation Management course please keep these points in mind.

(Refer Slide Time: 04:40)

The slide is titled "Program 10.08: Overloading operator new". It features a sidebar with navigation links for Module 10, Partha Pratim Das, Objectives & Outline, Memory Management in C and C++, and Overloading new & delete. The main content area contains the following C++ code:

```
#include <iostream>
#include <stdlib.h>
using namespace std;

void* operator new(size_t n) { // Definition of new
    cout << "Overloaded new" << endl;
    void *ptr;
    ptr = malloc(n); // Memory allocated to ptr
    return ptr;
}

void operator delete(void *p) { // definition of delete
    cout << "Overloaded delete" << endl;
    free(p); // Allocated memory released
}

int main() {
    int *p = new int; // calling overloaded operator new
    *p = 30; // Assign value to the location
    cout << "The value is : " << *p << endl;
    delete p; // calling overloaded operator delete
    return 0;
}

```

Annotations at the bottom right of the code area:

- * operator new overloaded
- * The first parameter of overloaded operator new must be size_t
- * The return type of overloaded operator new must be void *
- * The first parameter of overloaded operator delete must be void *
- * The return type of overloaded operator delete must be void
- * More parameters may be used for overloading
- * operator delete should not be overloaded (usually) with extra parameters

At the bottom of the slide, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" with page number "10".

Next, we look at something which is completely new in terms of C++, any corresponding concept of that in C did not exist. We have seen that operators can be overloaded and now we know that the Allocation operator new and the De-allocation operator delete are actually operators themselves, having corresponding operator functions and therefore, they can also be overloaded. So, in this here in first I show an example of the operator new, simple operator new being overloaded. So, if you look into the definition of operator new, you will see that the parameter it takes is a size_t parameter.

Now, this is what it actually expects is and just think about the corresponding call of this. So, this call will actually invoke this function. So, if you look at the call, the call does not exactly have a parameter - call has a type new int. So, if the call is saying apply operator new on type int, but the actual operator function expects a size_t, size_t is nothing but unsigned int. So, int C standard library size_t is typedef, it is a type aliased with unsigned

int. So, whenever its size underscore t (size_t) is written you will know that an unsigned int is meant and particularly it is size of something, most often it is a number of bytes of some structure and so on. So, size_t necessarily is expecting that n the parameter here for the operator function is necessarily the size of the int(sizeof(int)).

Now, you did not specify that sizeof. So, the trick is done by the compiler. The compiler knowing that you want int will actually convert; we will compute the size of that int and pass it as a parameter n. So, whenever you overload operator new, this size_t n must be the first parameter that must be specified without that the operator new will not work. And then what we have done we have provided a body for this operator new, here I have put a message just to make sure that this particular version of the overloaded operator is getting used, then I take a local pointer variable to void, then internally I am actually doing a malloc.

So, it feels as if it is operator new, but internally I am using a malloc to just allocate a raw amount of memory of the size of an int which n is now, and then we have just return that pointer because it as to return the pointer. And I have already explained that operator new will actually return you a pointer of the required type that is it will return you an int star.

In this case, it does not need a casting to be done, this is again a trick that the compiler will do. So, at the call time the compiler from the type, compute the size and pass it appropriately at the return time the compiler will appropriately change the type of the pointer and return a pointer of the required int type, but simply by writing this operator new function and writing a definition of that you are able to overload that.

So, you can see that operator new can be overloaded, the first parameter must be size_t in which the compiler will put the size of the type. Return type will have to be void* this you cannot change then it will not behave as an operator new, but you can actually add more parameters to this. Here, we have not added parameter, here we have just shown a different behavior by doing an allocation ourselves and by providing a message when this particular overloaded operator function works, but you can also add more parameters and pass those parameters from the new site.

Similarly, the operator delete can be overloaded as we have done it here again, the first parameter has to be void* which is the pointer that needs to be released and the return of

type has to be void because; obviously, delete cannot return any computation, but we can have a different implementation here from the default operator delete that exist. So, if you run this program you will find that at the time of allocation it prints overloaded new that is, this particular cout.

Then it prints the value from this point which is 30 and finally, it prints the outputs the overloaded delete message which comes from the overloaded delete operator. so, it clearly shows you that both new and delete have been overloaded and your behavior can be put.

One last point to remember here that when you overload delete, usually try not to overload it with extra parameters I cannot say that this is wrong according to C++, but there is a certain context in which only you can use a delete operator which is overloaded with more parameters than the void *p first parameter that it must have, and till we have come to explain those context you will not be, even if you overload a delete with extra parameters you will not be able to use them everything will be shown up as a compilation error.

(Refer Slide Time: 10:49)

Module 10
Partha Pratim Das
Objectives & Outline
Memory Management in C
Memory Management in C++
Memory Management in C++
Overloading operators

Program 10.09: Overloading operator new[]

```
#include <iostream>
#include <cstring>
using namespace std;

void* operator new [] (size_t os, char setv) { // Fill the allocated array with setv
    void *t = operator new(os);
    memset(t, setv, os);
    return t;
}

void operator delete[] (void *ss) {
    operator delete(ss);
}

int main() {
    char *t = new('#')char[10]; // Allocate array of 10 elements and fill with '#'
    cout << "p = " << (int) (t) << endl;
    for (int k = 0; k < 10; ++k)
        cout << t[k];
    cout << endl;
    delete [] t;
    return 0;
}
=====
p = 19421992
=====
```

* operator new[] overloaded with initialization
* The first parameter of overloaded operator new[] must be size_t
* The return type of overloaded operator new[] must be void *
* Multiple parameters may be used for overloading
* operator delete [] should not be overloaded (usually) with extra parameters

NPTEL MOOCs Programming in C++ Partha Pratim Das 11

So, here we show another example with the operator array new (operator [] new). So, this is also overloading. Now, I am trying to overload operator array new. So, this operator is to be used for allocation of an array of characters of 10 elements. So, naturally the first parameters still continue to be size_t which is the total number of bytes

that need to be allocated which can be computed here from this type and as well as the number of elements. So, `sizeof char[10]`, this will get passed as the `size_t` os parameter and this will be done by the compiler.

The interesting point to note is now you can see a second parameter the operator array new does not have a second parameter, but we have now overloaded it with a second parameter and just see the syntax in which the second parameter has to be used. So, this is the formal parameter here and the actual parameter is specified within a pair of parentheses right after the `new` keyword and before the type that we want to dynamically allocate.

So, this means that the character symbol `#` will be passed as a value of `setv` and then if you look into this code you will find that we are doing a `memset`. Let us look at the first line, here we are using the operator `new` function we could have used `malloc` also, but we are using operator `new` function to allocate the required amount of space which is `os` in this case, and we get that as a `void *` pointer and then using that pointer and using this `setv` parameter we actually fill up that buffer with the `setv` value.

So, we start from `t`, take a character and go `os` number of times because we know `os` number of bytes exist. These are basic `memset` function which exist in C, if you are not familiar with this function I would advise that you take a look at it separately in your C Programming. you can try it out, but this basically takes a pointer.

So, this is like, `t` is a pointer to this buffer which is got dynamically allocated. It has `os` number of locations and this will go and put the `setv` in this case the `#` character in each one of them. So, the difference with the other operator, `array new` and are overloaded `array new` will now be able to not only allocate a character array, but it will also be able to fill it up with the given character at the initial time. So, it does an initialization of a different way.

So, in this way you can think of various other overloading that you may need and accordingly you may have more and more parameters and behavior and those parameters, those extra parameters you will have to be passed as a comma separated list through the point that you show here.

Similarly, here the array delete is also overloaded here though we have not put any new functionality in that. we have simply called the delete operator function in this overloaded array delete. So, we have shown how to overload operator new in this case for initialization and again like a single item allocated operator new, operator array new also must have a first parameter which is size_t. It will always have a return type, which is void * and multiple parameters can be used for overloading an operator array delete, again like operator delete, if overloaded should not usually have extra parameters till we understand how those extra parameters are to be used.

(Refer Slide Time: 15:22)

The slide has a blue header bar with the title "Module Summary". On the left, there is a sidebar with a logo and a vertical list of module topics: Module 10, Partha Pratim Das, Objectives & Outcome, Memory Management in C, Memory Management in C++, and Overloading new & delete. At the bottom of the sidebar is a "Summary" button. The main content area contains a bulleted list of learning objectives:

- Introduced new and delete for dynamic memory management in C++
- Understood the difference between new, new[] and delete, delete[]
- Compared memory management in C with C++
- Explored the overloading of new, new[] and delete, delete[] operators

At the bottom of the slide, the footer includes the text "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and the number "12".

In summary we have introduced the new and delete operators for dynamic memory management in C++ these are in strong contrast from C where the malloc, free and other functions are actually standard library features, not the language features, here they are language features.

And we have carefully understood the differences between, different new operators and their corresponding delete operators and one basic principle that we have learnt is, whatever allocation we do the allocating function or operator that we use we must use the corresponding De-Allocating function or operator - malloc with free, operator new with operator delete, operator new with operator array delete and if I have done placement new then you should have no delete, and otherwise we are likely to get a whole lot of unpredictable problems.

We have also, at the end seen how to overload these new operators and delete operators, which is the advantage of having them as operators now. And we have shown how different functionality can be put in as desired by the user by overloading these operators.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 19
Classes and Objects

Welcome to module 11 of Programming in C++. We have already taken a loop into the procedural extensions of C++ over modules 5, 6 through 10. And we have looked at different better C features. Now from this module onwards, we will start discussing the core object-oriented features of C++, the concept of classes and objects.

(Refer Slide Time: 00:59)

The screenshot shows a presentation slide with a blue header bar containing the title 'Module Objectives'. Below the header, there is a sidebar on the left with a navigation menu. The menu includes 'Module 11' (selected), 'Partha Pratim Das', 'Objectives & Outline' (selected), and several other topics like 'Classes', 'Objects', 'Data Members', 'Copy', 'Assignment', 'Stack', 'Member Functions', 'Constructors', 'Assignment', 'Stack', 'this pointer', 'State of an Object', 'Copy', 'Assignment', 'Stack', and 'Summary'. At the bottom of the sidebar, it says 'NPTEL MOOCs Programming in C++'. On the right side of the slide, there is a large text area with a bullet point: '• Understand the concept of classes and objects in C++'. In the bottom right corner, there is a circular profile picture of Prof. Partha Pratim Das.

So, the objective of this module is to understand the concepts of class and objects in C++.

(Refer Slide Time: 01:07)

Module Outline

- Classes
- Objects
- Data Members of a class
- Member functions of a class
- this Pointer
- State of an Object

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, these are the items that we will walk through.

(Refer Slide Time: 01:12)

Classes

- A class is an implementation of a **type**. It is the only way to implement **User-defined Data Type (UDT)**
- A class contains **data members / attributes**
- A class has **operations / member functions / methods**
- A class defines a **namespace**
- Thus, classes offer **data abstraction / encapsulation** of **Object Oriented Programming**
- Classes are similar to structures that aggregate data logically
- A class is defined by **class keyword**
- Classes provide **access specifiers** for members to enforce **data hiding** that separates **implementation** from **interface**
 - **private** - accessible inside the definition of the class
 - **public** - accessible everywhere
- A class is a **blue print** for its instances (objects)

NPTEL MOOCs Programming in C++ Partha Pratim Das

Now, let me first give you a very basic overview of what is a class, and what is an object. We will slowly demonstrate, illustrate these through example, so that each and every point will become clear. As we say, a class is an implementation of a type. A statement

which will look or sound somewhat new to you totally because so far; so far as C was concerned, we either had built in types or we had types derived from the built-in type like arrays like structure or like pointers. But, now we will be in a position to implement a user defined data type all by ourselves and that will be one of the major lessons that we will try to take from the current module and the next couple of modules.

As we will see a class will contain data members or attributes, a class will have operations, member functions or methods; these are just alternate names of the same thing. We will see a class defines a name space that is once I define a class name, it becomes a surrounding property for all the data members and methods names that it contains. And in that way, a class will offer the data abstraction or the so-called encapsulation of object-oriented programming.

(Refer Slide Time: 03:24)

The slide has a blue header with the title 'Objects'. On the left, there's a sidebar with a logo and a navigation menu. The main content area contains a bulleted list of points about objects in C++.

Module 13
Partha Pratim Das
Objectives & Outcome
Classes
Objects
Data Members
 • Complex
 • Message
 • Stack
Member Functions
 • Complex
 • Message
 • Stack
 • this pointer
State of an Object
 • Complex
 • Message
 • Stack
Summary

Objects

- An **object** of a class is an **instance** created according to its **blue print**. Objects can be automatically, statically, or dynamically created
- A object comprises **data members** that specify its **state**
- A object supports **member functions** that specify its **behavior**
- Data members of an object can be accessed by "." (dot) operator on the object
- Member functions are invoked by "." (dot) operator on the object
- An implicit **this** pointer holds the address of an object. This serves the **identity** of the object in C++
- **this** pointer is implicitly passed to methods

NPTEL MOOCs Programming in C++ Partha Pratim Das 5

In terms of parallel with C programming, you can say that class is similar to structures that also aggregate data logically, but we will show how classes become different. To define class, C++ introduces a new keyword by the name class, and classes have different access specified; and finally, a class is a blue print and it can be instantiated for objects. So, objects are basically instances of a class; so, given one class, I can have

multiple instances of that class, a class tells us the blue print, tells us the format in which the data and the method should be organised for an object,

And every object will have a separate identity; it will have its own values of the data members that will specify the state in which the object will reside. It will support member functions, which will define the behaviour that it can offer; given an object we will be able to use the dot operator, the one that we had seen in terms of accessing components of structures will be able to use the same operator to access data members as well as methods of the object. And in addition, an object will have a special pointer known as this pointer; and this pointer will get implicitly passed to each and every method. This is just a very brief overview of what classes and objects are. I am sure at this very beginning it may not be making a whole lot of sense to you all. So, we will start with an example, and slowly illustrate each of these points over the next couple of slides.

(Refer Slide Time: 04:48)



Program 11.01/02: Complex Numbers: Attributes

Module 11 Partha Pratim Das Objectives & Outcome Classes Objects Data Members Complex Henceforth Stack Member Functions Classes Henceforth Stack this pointer State of an Object Complex Henceforth Stack Summary	<div style="display: flex; justify-content: space-between;"> <div style="width: 45%;"> C Program <pre>// File Name:Complex_object.c #include <stdio.h> typedef struct Complex { // struct double re, im; // Data members } Complex; int main() { // Variable n1 declared, initialized Complex n1 = {4.2, 5.3}; printf("%d %d", n1.re, n1.im); // Use return 0; } return 4.2 5.3 </pre> <ul style="list-style-type: none"> ★ struct is a keyword in C for data aggregation ★ The struct Complex is defined as composite data type containing two double (re, im) data members ★ struct Complex is a derived data type used to create Complex type variable n1 ★ Data members are accessed using <code>."</code> operator ★ struct only aggregates </div> <div style="width: 45%;"> C++ Program <pre>// File Name:Complex_object,c++.cpp #include <iostream> using namespace std; class Complex { public: // class double re, im; // Data members }; int main() { // Object n1 declared, initialized Complex n1 = {4.2, 5.3}; cout << n1.re << " " << n1.im; // Use return 0; } return 4.2 5.3 </pre> <ul style="list-style-type: none"> ★ class is a new keyword in C++ for data aggregation ★ The class Complex is defined as composite data type containing two double (re, im) members ★ class Complex is User-defined Data Type (UDT) used to create Complex type object ★ Data members are accessed using <code>."</code> operator ★ class aggregates and helps to do re building a UDT </div> </div> <div style="text-align: right; margin-top: -20px;">  </div>
--	--

NPTEL MOOCs Programming in C++

So, let us consider a simple definition of a complex number. We had taken this example earlier also, where we can actually use a structure which has two data members, two members re and im designating the two components of a complex number and that is defined as a structure and we typedef that, alias that to be the name Complex. So, once we had done that then we can define variables of that structure and put some initial

values to this component of the structure. So, `re` will become 4.2, and `im` will become 5.3. So, if we now print the two components of this complex number in one, it will print out as 4.2 and 5.3. This is what you already know; this is what is available in C.

If I want to write this very similarly in C++, I will change and write this as class `complex`. So, we were doing a struct `complex` and then we are doing aliasing with the `typedef`, now I simply write a class `complex` and put the member definitions within that. And rest of the code can be written in a very similar manner only difference being here, we are using `printf` and here we are using `cout` as we have seen in case of the C++ we use the streaming operators for doing this.

Now, let us see what difference it has actually made. So, `struct` is a keyword in C, `class` is a new keyword in C++. Rest of what we have done are very similar between the two approaches of defining and aggregation of two components of the real component and the imaginary component of a complex number, either through a struct in C or through a class in C++. So, the only contrast that we will slowly bring out now is, while `struct` will allow us to only aggregate, put the two components together, refer to them together as we are referring to `n1` as a as a pair of two double numbers 4.2 and 5.3 designating a complex the class also does the same thing. But class we will see will do a lot more things than what `struct` can do.

(Refer Slide Time: 07:42)

C Program	C++ Program
<pre> // File Name:Rectangle_object.c #include <stdio.h> typedef struct { // struct Point int x; int y; } Point; typedef struct { // Rect uses Point Point TL; // Top-Left Point BR; // Bottom-Right } Rect; int main() { Rect r = {{0,2}, {5,7}}; // r.TL.x == 0; r.TL.y == 2 // r.BR.x == 5; r.BR.y == 7 // Members of structure r accessed printf("(X Y) (X Y)\n", r.TL.x, r.TL.y, r.BR.x, r.BR.y); return 0; } </pre>	<pre> // File Name:Rectangle_object_c++.cpp #include <iostream> using namespace std; class Point { public: // Data members int x; int y; }; class Rect { public: // Rect uses Point Point TL; // Top-Left Point BR; // Bottom-Right }; int main() { Rect r = {{0,2}, {5,7}}; // r.TL.x == 0; r.TL.y == 2 // r.BR.x == 5; r.BR.y == 7 // Rectangle Object r accessed cout << "(" << r.TL.x << " " << r.TL.y << " " ") (" << r.BR.x << " " << r.BR.y << ")\n"; return 0; } </pre>
<p>• Data members of user-defined data types</p>	

NITTEL MOOCs Programming in C++ Partha Pratim Das 7

So, let us take a little different and bigger example. Here we are trying to define a rectangle and this is a special kind of rectangle which will say is isothetic rectangle in the sense that this rectangle has its size parallel to the axis x and y-axis. So, if I just specify with two corners, diagonally opposite corner, the top left and the bottom right corners of the rectangle, then the rectangle is fully specified. For doing this, first we define a structure which gives us points as an aggregation of two coordinates x and y and then we take two points and let them designate that two diagonally opposite points of a rectangle. And once this has been done then I can define a rectangle here, rectangle r. You can see the notation the {0, 2} here first, within the first pair of curly braces mean the point top left, {5, 7} mean the point bottom right. And both of them together mean the whole rectangle.

So, in the comments, you can see that actually if I specify 0, 2 as the first point, then I am actually specifying r being the name of the rectangle r.TL is a top left point, dot x (.x) is x coordinate of that which gets the value 0. Similarly, r.TL.y, which is a y component of the top left point, gets the value 2. So, through this initialization all these values are being set and then I can print them out. Similarly, I can write the whole thing using class. The difference is being exactly as it was before. I can initialize it in the similar way and I can print it out using cout. So, this is just showing another example of using the class.

Here the data member in Rect - the rectangle, have actually not of any basic type, but they are user defined data types, they are class objects themselves, they are instances of the point class that I have already defined earlier.

(Refer Slide Time: 10:19)

C Program	C++ Program
<pre>// File Name:Stack_object.c #include <stdio.h> typedef struct Stack { // struct Stack char data [100]; int top; } Stack; // Codes for push, pop, top, empty int main() { // Variable s declared Stack s; s.top = -1; // Using stack for solving problems return 0; }</pre>	<pre>// File Name:Stack_object.cpp #include <iostream> using namespace std; class Stack { public: // class Stack char data [100]; int top; }; // Codes for push, pop, top, empty int main() { // Object s declared Stack s; s.top = -1; // Using stack for solving problems return 0; }</pre>

• Data members of mixed data types

Let us take third example of a stack, which is something we have been discussing earlier too. So, in the stack, we have a mix combination of mixed data types, we have a character array to keep the elements of the stack, it is the stack of characters, we have a top marker which is an index on the stack which shows where the top actually exists. So, I can define a stack variable here or here; and obviously, before I start using the stack, I need to make sure that the stack is empty which is designated by the top index being minus 1 (-1). So, I initialize the top index to -1, and then I can use the stack either, whether it is defined through struct or is define through class by using different algorithms for solving different problems.

So, these are different examples, where you can see that, we show that the basic components of a class, that is a class as a class keyword, it has a name which is the name and identifier, it has a number of data members. Each data member is defined as in struct as a variable declaration style and using the name of the class, we can declare variables of the class types these are known as instances. So, this is an instance or this is what is

called an object. So, s is an object of the class stack. And once we have that then with that object, we can use the data member, the data member here is top using the dot member notation. So, we first show that a class is an aggregate which can put together one or more data members and allows us to instantiate the objects of that class or define variables of that type using the C++ mechanism.

(Refer Slide Time: 12:41)

The slide has a blue header with the title 'Classes'. On the left, there is a vertical navigation menu with the following items: Module 11, Partha Pratim Das, Objectives & Outline, Classes, Objects, Data Members, Complex Message Stack, Member Functions, Complex Message Stack, this pointer, State of an Object, Complex Message Stack, and Summary. The main content area contains a bulleted list of six points about classes. At the bottom right, there is a circular profile picture of Partha Pratim Das.

- A class is an implementation of a **type**. It is the only way to implement **User-defined Data Type (UDT)**.
- A class contains **data members / attributes**.
- A class defines a **namespace**.
- Thus, classes offer **data abstraction / encapsulation** of **Object Oriented Programming**.
- Classes are similar to structures that aggregate data logically.
- A class is a **blue print** for its instances (objects).

So, just to recap, we have seen that class is the implementation of a type we will see this more. We have seen three attempts to do three types a complex type, a rectangle and point type and a stack type. We have shown that the class will contain data members, it defines a name space that is when I say that my I am defining a complex, my all data members of complex actually have a name, which is qualified by the complex, the class name itself. And it is aggregating the data logically.



Objects

Module 11
Partha Pratim Das

- [Objectives & Outline](#)
- [Classes](#)
- [Objects](#)
- [Data Members](#)
- [Complex Rectangle Stack](#)
- [Member Functions](#)
- [Complex Rectangle Stack](#)
- [this pointer](#)
- [State of an Object](#)
- [Complex Rectangle Stack](#)
- [Summary](#)

● An **object** of a class is an **instance** created according to its **blue print**. Objects can be automatically, statically, or dynamically created

● A object comprises **data members** that specify its **state**

● Data members of an object can be accessed by “.” (dot) operator on the object

NPTEL MOOCs Programming in C++ Partha Pratim Das

In terms of objects instances, we have shown that for every type of class, every three classes that we have defined, we can define in different instances of those or objects of those and data members are accessed through the “.” dot operations.

(Refer Slide Time: 13:43)



Program 11.07/08: Complex Numbers: Methods

Module 11
Partha Pratim Das

- [Objectives & Outline](#)
- [Classes](#)
- [Objects](#)
- [Data Members](#)
- [Complex Rectangle Stack](#)
- [Member Functions](#)
- [Complex Rectangle Stack](#)
- [this pointer](#)
- [State of an Object](#)
- [Complex Rectangle Stack](#)
- [Summary](#)

C Program	C++ Program
<pre>// File Name:Complex_func.c: #include <stdio.h> #include <math.h> typedef struct Complex { double re, im; } Complex; // Norm of Complex Number = global fn. double norm(Complex c) { return sqrt(c.re*c.re + c.im*c.im); } // Print number with Norm = global fn. void print(Complex c) { printf("(%.1f+%.1fi) = ", c.re, c.im); printf("(%.1f)", norm(c)); // Call global } int main() { Complex c = { 4.2, 5.3 }; // Call global fn. with 'c' as param print(c); return 0; } ***** 14.200000+j5.300000] = 6.762396</pre>	<pre>// File Name:Complex_func_c++.cpp: #include <iostream> #include <cmath> using namespace std; class Complex { public: double re, im; }; // MEMBER FUNCTIONS / METHODS // Norm of Complex Number = method double norm() { return sqrt(re*re + im*im); } // Print number with Norm = method void print() { cout << "[" << re << "+" << im << "i" << "] = " cout << norm(); // Call method } }; // End of class Complex int main() { Complex c = { 4.2, 5.3 }; // Invokes method print of 'c' c.print(); return 0; } ***** 14.2+5.3i = 6.7624</pre>

Partha Pratim Das

Now, so far what we have seen is something which is strongly in parallel to what we could do as structures. And now we are going to make the first major distinction or the first major step to move away from what we could do in structure, in terms of the class

definition. So, please follow this very carefully. We are back to the complex examples. So, this part is common, you have the data members here in terms of the structure definition, we have the data members the same way in terms class definition. Now, if I have such a complex number in C, I am looking at the left side of the slide, then I can define a number of function say I define a function norm which can takes such a complex number C as in here and define its norm. You all know how the norm is computed this sum of the square of the real and the imaginary parts. And then they take a square root of that sum, you get the norm of the complex number or we can write another function to print the complex number in the real plus j imaginary component kind of notation print it's norm value and so on.

So, these functions can be written along with the structure complex type that I have already defined in C. And these are all as we know are C functions or more commonly global functions, and then I can use to print and if I do that the complex number will be printed out. So, if you just want to take a look as to, how it what it prints, this is what it prints. So, given the complex number is 4.2, 5.3, it prints that $4.2 + j 5.3$ the norm of that is 6.7624.

Now, lets us look at the C++ side carefully. Here I am also defining the norm function, but with the difference. In the struct, case of struct, the struct definition is separate, my function definition is separate, but here my class definition, this is my whole of class definition of complex, and my function is a part of the class definition. And such functions will be called, quite naturally will be called member functions like `re` is a member of the class `complex`, we call it is a data member, similarly “`double norm()`” this function is also a member of the class `complex`. And it is called a member function or a method. You can also see that the other function `print` is also being contained within the definition of the class and `print` is yet another member function.

So, this is something member function is a completely new concepts for C++, no parallel of this exist in C. And with this member function, now we can say that my object instances, that is given the class `complex` C is an instance of this class, now my object instances, say this instance C can use the method in this notation. Earlier, you had seen the notation in terms of data member only, that is “`c.re`” is something we had seen which

means I am referring to the `re` data member of the complex number `C`. But now I am writing “`c.print`” which means that for the object `c`, for the instance `c`, I am making use or I am invoking the method `print()`. That is method `print` whatever it is suppose to do, it will do assuming that these data members have the values that the `c` object has and accordingly it will work, it will print out. Similarly, if I write `c.norm()` and invoke, this will invoke the other member function `norm()`, and the norm will also behave with the real and imaginary components coming from the value of `c`.

This is the completely new concepts and this is what the member functions would do. So, we are contrasting that as in C, if we define something as struct, every operation we need to do with it, we need to be done through some global functions that anybody and everybody can see and use. Whereas in C++, the class itself would have, could define a number of member functions or method which the object can invoke as and when necessary to do some operations, and this is what is known as the behaviour of the object as we will see slowly.

(Refer Slide Time: 19:09)

Program 11.09/10: Rectangles: Methods	
Module 11	
Partha Pratim Das	
Objectives & Outline	
Classes	
Objects	
Data Members	
Complex Functions	
Message Passing	
Member Functions	
Complex Rectangle Class	
this pointer	
State of an Object	
Complex Message Passing	
Summary	

```

Using struct
#include <iostream>
using namespace std;

typedef struct {
    int x; int y;
} Point;
typedef struct {
    Point TL; // Top-Left
    Point BR; // Bottom-Right
} Rect;

// Global function
void computeArea(Rect r) {
    cout << abs(r.TL.x - r.BR.x) *
        abs(r.BR.y - r.TL.y);
}

int main() {
    Rect r = { { 0, 2 }, { 6, 7 } };
    // Global fn. call
    computeArea(r);
    return 0;
}
26

Using class
#include <iostream>
using namespace std;

class Point { public:
    int x; int y;
};

class Rect { public:
    Point TL; // Top-Left
    Point BR; // Bottom-Right
};

// Method
void computeArea() {
    cout << abs(TL.x - BR.x) +
        abs(BR.y - TL.y);
}

int main() {
    Rect r = { { 0, 2 }, { 6, 7 } };
    // Method invocation
    r.computeArea();
    return 0;
}
26

```

NPTEL MOOCs Programming in C++

Partha Pratim Das

So, let us look at some more examples. Let us again bring back the rectangle point part, this part you have already seen, this part you have already seen. Here we are writing a function for C using struct, this is a global function, which computes the area of the

rectangle. The formula for computing the area is straightforward, I will not get deeper into that, but this is a global function computeArea(), which takes a rectangle as a parameter and computes the area and this is how it works. In contrast, in the C++ class, this is what my rectangle class, this is what my rectangle class is my method, my computeArea() method is a member functions that is a part of the class.

So, when I have a rectangle r object, I invoke the method or member function as in the same using the same dot notation, and it means that the compute area will work, this will work assuming that r as instantiated here is the object for which it is working. So, when it refers to TL.x, it actually refers to the TL.x of the r object, which has invoked this particular method. This is more as in C, we say this is function call, we continue to say it is a function call in C++ as well, but these when you use the member functions of different objects, you often would say that you are invoking the method of an object. A method or a member function, which is available as a part of its class definition, so that is the core idea of methods in C++.

(Refer Slide Time: 21:16)


**Program 11.11/12: Stacks:
Methods**

Module 11 Partha Pratim Das Objectives & Outline Classes Objects Data Members Complex Rectangle Stack Member Functions Complex Rectangle Stack this pointer State of an Object Complex Rectangle Stack Summary	<div style="background-color: #e0e0ff; padding: 5px; margin-bottom: 5px;"> Using struct </div> <pre>#include <iostream> using namespace std; typedef struct Stack { char data_[100]; int top_; } Stack; bool empty(const Stack& s) { return (s.top_ == -1); } char top(const Stack& s) { if (empty(s)) return -1; else return s.data_[s.top_]; } void push(Stack& s, char x) { s.data_[++(s.top_)] = x; } void pop(Stack& s) { --(s.top_); } int main() { Stack s; s.top_ = -1; char str[10] = "ABCDE"; int i; for (i = 0; i < 5; ++i) s.push(s, str[i]); cout << "Reversed String: "; while (!empty(s)) { cout << top(s); s.pop(); } return 0; } </pre> <div style="background-color: #e0e0ff; padding: 5px; margin-top: 5px;"> Using class </div> <pre>#include <iostream> using namespace std; class Stack { public: char data_[100]; int top_; // METHODS bool empty() { return (top_ == -1); } char top() { return data_[top_]; } void push(char x) { data_[++top_] = x; } void pop() { --top_; } }; int main() { Stack s; s.top_ = -1; char str[10] = "ABCDE"; int i; for (i = 0; i < 5; ++i) s.push(str[i]); cout << "Reversed String: "; while (!s.empty()) { cout << s.top(); s.pop(); } return 0; }</pre> <div style="text-align: right; margin-top: 10px;">  Partha Pratim Das </div>
--	---

NPTEL MOOCs Programming in C++

You could take a look. We will not go very detailed into this you could take your time when you study this more, this presentation more. You can see that this is a complete example of a stack which, these has the data. These are the four operations of stack:

empty, top, push and pop, in C given as global functions. We use the instance of a stack here, initialize it is a top marker, use it and for a given string, we push each and every character of that string one after the other into the stack, and then we keep on popping them till the stack gets empty. As you know this is the standard algorithm to reverse a string. So, this code will reverse the string using this global function.

And now we show that we can do the same thing as we have been showing using class complex, as we have shown using class rectangle. We show that in case of class stack, I could have the data, and I could make all of these stack operations as we need to do them into member functions and method a part of the class. And as soon as we make them as part of the class, then we use them just look at the difference. If you do push here you need to say what is the stack and then what are you pushing here, you say the stack is s. So, you are invoking the push method for the stack object and so you are just saying what you pushing here. Similarly, for checking empty you pass the stack s to the global function empty, here empty is a member function. So, you take for the stack s you simply take the method invoke the method empty which will us this method to find out if the stack s has a top which is equal to minus 1.

Similarly, top s is invoked here as s.top(); pop s is invoked here as s.pop(). So, you can see that it is in contrast to using global functions, now this method invocation or member functions are allowing us to put together the data as well as the operations, as well as the methods or member all that I need to do with the data put to give them together into one unified bundle. And that is what gives a more proper, more accurate encapsulation or aggregation as object oriented programming would often ask for.

(Refer Slide Time: 24:03)

The slide has a blue header with the title 'Classes'. On the left, there is a vertical navigation menu with items: Module 11, Partha Pratim Das, Objectives & Outline, Classes (which is the current page), Objects, Data Members, Member Functions, this pointer, State of an Object, and Summary. The main content area contains a bulleted list:

- A class has **operations / member functions / methods**
- A class defines a **namespace**
- Thus, classes offer **data abstraction / encapsulation** of **Object Oriented Programming**

At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

So, with this, what we achieve that, the class has operations and member functions methods and it offers a data abstraction or encapsulation as in OOP, we will obviously, have more to understand this.

(Refer Slide Time: 24:20)

The slide has a blue header with the title 'Objects'. On the left, there is a vertical navigation menu with items: Module 11, Partha Pratim Das, Objectives & Outline, Classes (which is the current page), Objects, Data Members, Member Functions, this pointer, State of an Object, and Summary. The main content area contains a bulleted list:

- An **object** of a class is an **instance** created according to its **blue print**. Objects can be automatically, statically, or dynamically created
- A object supports **member functions** that specify its **behavior**
- Member functions are invoked by '.' (dot) operator on the object

At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'. There is also a circular profile picture of Partha Pratim Das.

And in terms of objects we know that member functions are defining its behaviour; as we have just seen that in stack the member functions like push, pop, empty, top defines all the LIFO behaviour that the stack needs to do. The advantage being that when we use C and we are using global functions, the global functions do not know about the stack structure, the stack structure has no idea about what global functions, I might pass it into. But when I do a class and put them together the data member and the methods that work with the data member are completely closely tied together, so you deal with them necessarily together and we will see particularly after access specification how they become completely tightly bound to give us a more complete data type in C++.

(Refer Slide Time: 25:08)

Program 11.13: this Pointer

- An implicit this pointer holds the address of an object
- this pointer serves as the **identity** of the object in C++
- Type of this pointer for a class X object: X * const this;
- this pointer is accessible *only in methods*

```
#include <iostream> using namespace std;
class X { public: int a1, a2;
    void f(int k1, int k2) {
        a1 = k1; // Implicit access w/o 'this' pointer
        this->a2 = k2; // Explicit access w/ 'this' pointer
        cout << "Id = " << this << endl; // Identity (address) of the object
    }
    int main() {
        X a;
        a.f(2, 3);
        cout << "addr = " << &a << endl; // Address (identity) of the object
        cout << "a.a1 = " << a.a1 << " a.a2 = " << a.a2 << endl;
        return 0;
    }
}
-----
Id = 0024F918
Addr = 0024F918
a.a1 = 2 a.a2 = 3
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

Now, the next concepts that we need to understand, introduces here is a concept of this pointers. This pointer, “this” is actually a keyword in C++, and this is a name. And it is an implicit pointer that holds an address of an object. So, if I am talking about an object, the object can refer to itself, its own identity, its own address within the methods of the object can be referred to as this. And this pointer has an interesting signature. So, you have already seen the const pointer, concepts earlier, so you can easily read this signature of how this pointer is implicitly defined. So, for a class X, the “this” pointer of its object will be “x * const this”, which “x *” tells it that this is a pointer to a class type of a object and const after this star of the pointer type tells us that this is a constant pointer that is

you cannot change the value of this pointer, which is what make sense because we are saying that this is the address of an object. So, it is accessible in terms of different methods.

So, here I just show a couple of examples. This does not do anything fruitfully meaningful, but this is just for illustration that; X is a class, it has two members. And f is one function, which takes two parameters and sets them to two data members of the object. And we can refer to the data member by either directly as m1 or I can refer to it by this pointer m2, because the implicitly what it means that, if I am talking about a object then I have a pointer which is a pointer to this object. So, when I am in f, I have as this, I have the value of this pointer through which I can refer to m1 and m2 in this object. So, if you go through this code, you will able to see that this pointer is actually carrying the address. So, in the main code, we have printed the address; and in the function f, we have printed the value of this pointer and you can see that they are identical. So, this pointer actually carries the address of the object.

(Refer Slide Time: 27:49)

The slide has a blue header with the title 'this Pointer'. On the left, there's a sidebar with navigation links: Module 13, Partha Pratim Das, Objectives & Outline, Classes, Objects, Data Members, Complex Message Stack, Member Functions, Complex Message Stack, this pointer, State of an Object, Complex Message Stack, and Summary. The main content area has two sections: 'this pointer is implicitly passed to methods' and 'Use of this pointer'. The first section shows source code for a class X with a method f(int, int) and binary code for calling it. The second section discusses distinguishing member from non-member functions and provides an example of explicit use with a DoublyLinkedListNode class.

In Source Code	In Binary Code
<code>class X { void f(int, int); ... }</code>	<code>void X::f(X * const this, int, int);</code>
<code>X a; a.f(2, 3);</code>	<code>X::f(&a, 2, 3); // &a = this</code>

- this pointer is implicitly passed to methods

```

class X { public: int m1, m2;
          void f(int k1, int k2) {
              m1 = k1;           // this->m1 (member) is valid; this->k1 is invalid
              this->m2 = k2; // m2 (member) is valid; this->k2 is invalid
          }
      };
  
```

- Use of this pointer
 - Distinguish member from non-member


```

class X { public: int m1, m2;
          void f(int k1, int k2) {
              m1 = k1;           // this->m1 (member) is valid; this->k1 is invalid
              this->m2 = k2; // m2 (member) is valid; this->k2 is invalid
          }
      };
  
```
 - Explicit Use


```

// Link the object
class DoublyLinkedListNode { public: DoublyLinkedListNode *prev, *next; int data;
                           void append(DoublyLinkedListNode *x) { next = x; x->prev = this; }
                         }
...
// Return the object
Complex& inc() { ++re; ++im; return *this; }
  
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, in here, I just, it is usually optional to use this pointer when you are accessing different data members or methods of a class, but you can use them to distinguish data member from other variables like k1, k2 here. But there are some instances, some

situations where it becomes very necessary and we have just put in two examples here. For examples, if you have a doubly linked list and you want to put in a node which inserts a node after a given node, which will forward link to the next node and backward link to the earlier node, you will need to use the address of the node that you are putting in. Or if you are returning an object, then you will need to refer to the object that you are returning itself, we will see more examples of these later on. So, I will just like you to try this out, but we will detail it further when we do further examples on this.

So, with this we have learnt what is the concept of a class and its corresponding object. We have also learnt about data members of a class, and of the object corresponding object. We have learnt about the methods, which can be defined for a class and invoked for an object. And we have seen that every object has an identity, which can be captured in “this” pointer of the methods of that object, and that carries an address which is the address of the object. At this point, I would also like to just touch up on that, in C++, the objects do not have any other separate identity. So, this pointer or the address of the object is taken to be the address everywhere, which is different from what some of the other object-oriented languages do.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 20
Classes and Objects (Contd.)

Welcome to part-2 of module 11, of Programming in C++. In the earlier part of this module, we have seen the basic concepts of classes and instances of classes as objects. We have understood, what are data members and methods. We have discussed three examples particularly of complex numbers; of rectangle objects with points and of a stack. We have also understood “this” pointer for identifying an object.

In the remaining part we would briefly discuss what is known as the State of an Object.

(Refer Slide Time: 00:58)

The slide has a blue header bar with the title "State of an Object: Complex". The sidebar on the left contains the following navigation links:

- Module 11
- Partha Pratim Das
- Objectives & Outline
- Classes
- Objects
- Data Members
- Complex Rectangle Stack
- Member Functions
- Complex Rectangle Stack
- this pointer
- State of an Object
- Complex Rectangle Stack

The main content area contains two bullet points and some code:

- The state of an object is determined by the combined value of all its data members. Consider class Complex:

```
class Complex { public:  
    double re_, im_; // ordered tuple of data members decide the state at any time  
  
    double get_re() { return re_; }  
    void set_re(double re) { re_ = re; }  
    double get_im() { return im_; }  
    void set_im(double im) { im_ = im; }  
}
```

A hand-drawn diagram shows a box labeled "Complex" containing a "re" field with value "4.2" and an "im" field with value "5.3".
- A method may change the state:

```
Complex c = {4.2, 5.3};  
// STATE 1 of c = {4.2, 5.3}  
  
c.set_re(6.4);  
// STATE 2 of c = {6.4, 5.3}  
  
c.get_re();  
// STATE 2 of c = {6.4, 5.3} // No change of state  
  
c.set_im(7.8);
```

This is a notion which is derived from the way the object-oriented paradigm applies in the contexts of C++. We say that the state of an object is determined by the combined value of all its data members.

In simple terms, let us say that if we are going back to the complex example. So, this data part is same of course, we have some additional methods just for the purpose of demonstration. So, these data members what do they say? They say that if I have a complex number, if I can just draw it out, then a complex number is, it has a ‘re’ component then and an ‘im’ component. So, this is a complex number. So, if I say c1 is defined initialized to be 4.2, 5.3 then I can say this is c1 and this is 4.2 and this is 5.3.

So, we say the state, notion of state is this is a double value; this also has a double value. So, I know that I can have any double value as the value of re. Similarly, I can have any double value as a value of im. So, every possible pair of double value that I can put to re and to im will denote a different complex number. So, if I change in c1, if I change this value or if I change this value or if I change both the values. So, we will say that c1 has acquired a different state.

As long as the data members of an object maintain a certain combination of values, we say it is in one state and as soon as any one of the data members change in its value, we say it is in a different state. So, it is looked at this way that finally, in programming, it is about deciding on what state the object is in and with the use of methods, what is the next state the object could get into. We will look at all these dynamics subsequently more, but this is just to show you that, what is the notion of state in depth?

To show that we just make a use of four methods here, just look at them closely, `get_re`; basically, reads returns the component re. So, if c1 is 4.2, 5.3 and if I invoke `c1.get_re`, I will obviously get 4.2. Similarly, if I do `get_im`, it will return me 5.3 and the other two these are set. So, basically if I pass a value as re to the `set_re` method, then it will set that value into the `re` component of the object on which I have called `set_re`. So, these are commonly called get set method, we will look at them more. So, with that let us see, what are the different states that I can be into? So, this is the initial one, it is initialized with 4.2, 5.3. So, state 1 of c is 4.2, 5.3, since there are two data members the state is defined in terms of doublet, in terms of a pair of numbers here.

Then, if I do `c.set_re(6.4)`, then the 4.2 changes to 6.4. So, I have a new state two which is 6.4, 5.3 here, that is state 2. Now, in this let us suppose I invoke `c.get_re`, which will

read the re component of the c object which is basically, now 6.4. So, it will return 6.4, but you can note that, there is no change in the re or im components of the object. Therefore, you will conclude that there is no change in the state, so it continues to be in state 2. But if I do set_im(7.8), then naturally my state will change because now the object will become (6.4, 7.8). So, in this way as different operations are performed on an object, it goes through different states and we always say that the data members in a way remember the state of an object.

(Refer Slide Time: 06:03)

State of an Object: Rectangle

- Consider class Point and class Rect:

```

Data members of Rect class: Point TL; Point BR; // Point class type object
Data members of Point class: int x; int y

Rectangle r = {{0, 5}, {5, 0}}; // Initialization
// STATE 1 of r = {{0, 5}, {5, 0}}
{ r.TL.x = 0; r.TL.y = 5; r.BR.x = 5; r.BR.y = 0 }

r.TL.y = 9;
// STATE 2 of r = {{0, 9}, {5, 0}}

r.computeArea();
// STATE 2 of r = {{0, 9}, {5, 0}} // No change in state

Point p = {3, 4};
r.BR = p;
// STATE 3 of r = {{0, 9}, {3, 4}}

```

NPTEL MOOCs Programming in C++ Partha Pratim Das

These are more examples for you to practice. This is an example showing a rectangle, its corner points and as we changed different coordinates of the corner points or we compute ratio, we show how the state of object is changing.

(Refer Slide Time: 06:22)

Module 11
Partha Pratim Das
Objectives & Outline
Classes
Objects
Data Members
Complex Rectangle Stack
Member Functions
Complex Rectangle Stack
this pointer
State of an Object
Complex Rectangle Stack
Summary

State of an Object: Stack

- Consider class Stack:

```
Stack s;  
// STATE 1 of s = {{?, ?, ?, ?, ?}, ?} // No data member is initialized  
s.top_ = -1;  
// STATE 2 of s = {{?, ?, ?, ?, ?}, -1}  
s.push('b');  
// STATE 3 of s = {{'b', ?, ?, ?, ?}, 0}  
s.push('a');  
// STATE 4 of s = {{'b', 'a', ?, ?, ?}, 1}  
s.empty();  
// STATE 4 of s = {{'b', 'a', ?, ?, ?}, 1} // No change of state  
s.push('t');  
// STATE 5 of s = {{'b', 'a', 't', ?, ?}, 2}  
s.top();  
// STATE 5 of s = {{'b', 'a', 't', ?, ?}, 2} // No change of state  
s.pop();  
// STATE 6 of s = {{'b', 'a', 't', ?, ?}, 1}
```

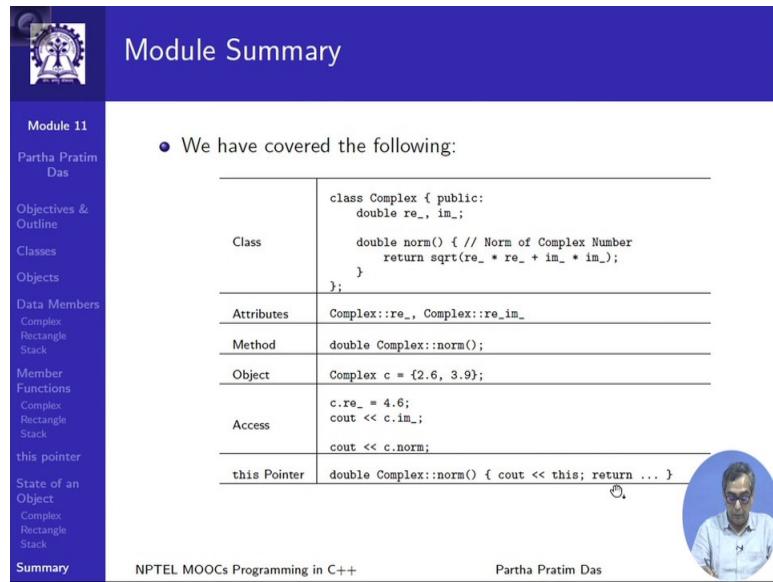
NPTEL MOOCs Programming in C++ Partha Pratim Das

There is another example on stack. So, in stack what did we have? We have one data array and the indicated top. So, the state will comprise the whole array. So, if the data array has size 5, then it has a 5 tuple denoting all the characters that, this data array has and then another component top, which is the value of top. So, these together will give me the state of s and these are all question marks because at the beginning nothing has been initialized. So, I do not know what state the object is in.

But, then as I perform initialization to the top, this becomes minus 1. So, I get some state, but I still do not know, what is the state of the array I push b. So, the first character becomes b this gets incremented to 0, my state changes, I push a, 'b' 'a' this changes. When I check for empty it is not empty and it returns me false and it does not change the array or the top. So, there is no change of state and so on.

So, if you just proceed to follow this you will see that with the operations of push and pop, we will actually be changing the state, whereas with top and empty we will not be changing the state and at any point the stack can be described in terms of the state of its array and the state of top marker. So, you would have frequently heard about objects having states. So, this is the basic meaning of the state as you go forward, we will discuss more about the behavior of on object.

(Refer Slide Time: 08:13)



The slide is titled "Module Summary" and features a sidebar with navigation links for Module 11, including "Objectives & Outline", "Classes", "Objects", "Data Members", "Member Functions", "this pointer", and "Summary". The main content area contains a bulleted list: "We have covered the following:" followed by a table. The table has six columns: Class, Attributes, Method, Object, Access, and this Pointer. Each column contains code snippets. A small circular portrait of Partha Pratim Das is on the right.

Class	class Complex { public: double re_, im_; double norm() { // Norm of Complex Number return sqrt(re_ * re_ + im_ * im_); } };				
Attributes	Complex::re_, Complex::re_im_				
Method	double Complex::norm();				
Object	Complex c = {2.6, 3.9};				
Access	c.re_ = 4.6; cout << c.im_; cout << c.norm;				
this Pointer	double Complex::norm() { cout << this; return ... }				

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, with this we will close on the module 11. In module 11, we have covered the following we have understood the basic concept of a class with data members and methods. We have seen that the attributes or data members can be named in the name space of complex. So, re, the name of re_ underscore actually is complex::re and so on.

The method is also named similarly in the name space of complex. So, a method norm has a name complex::norm(). The objects are instantiations of the classes and they can be initialized when they are instantiated, axis is done in terms of using the dot operator and there is a special “this” pointer which identifies every object by its own address, which can be used in different methods.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

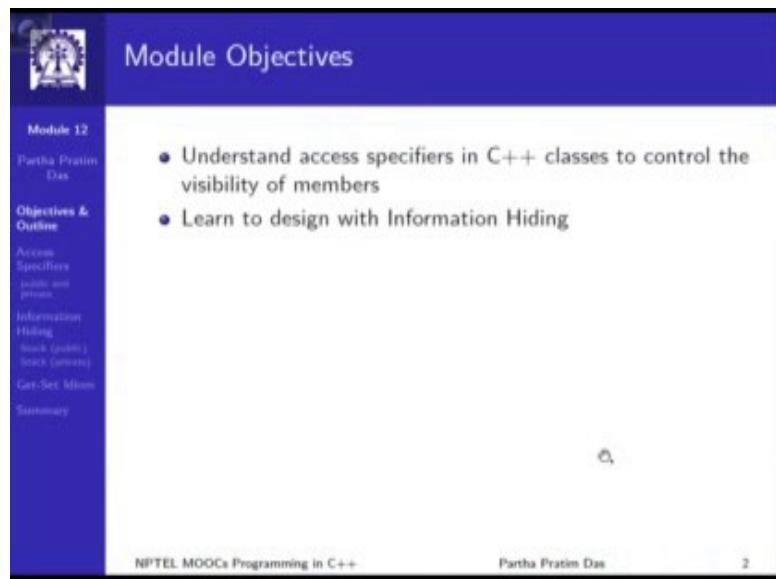
Lecture - 21
Access Specifiers

Welcome to Module 12 of programming in C++. In module 11, we have been introduced the basic concept of classes and objects. We have understood that using the class keyword in a style, which is very similar to the way we write struct in C. We can define aggregations of one or more data members.

We have also learned that along with the data members, the class also allows us to write functions. Which can work with those data members these functions, these special functions are called member functions or methods. And they could be invoked with a dot operator much in the same way that the data members are accessed. We have also seen that an object as an instance of a class every object as a unique identity, which is the address of that object and that address itself can be accessed within any member function of the class given the instance of the object has a special pointer known as this pointer.

So, we will continue on this and today we will focus on access specifiers or more specifically we would take a closer look into how class in C++ can be used to define better object encapsulation in object-oriented programming.

(Refer Slide Time: 02:05)



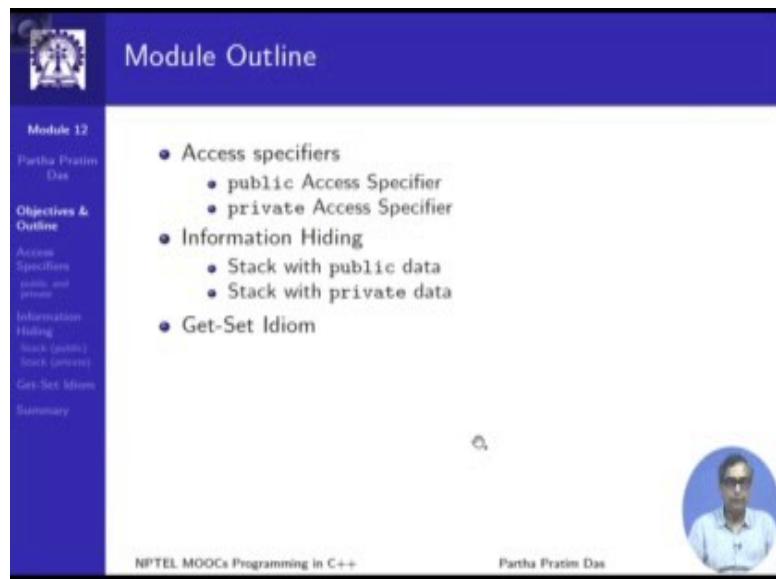
The slide has a blue header bar with the title "Module Objectives". On the left, there is a vertical sidebar with a navigation menu. The menu items include: Module 12, Partha Pratim Das, Objectives & Outline, Access Specifiers (public and private), Information Hiding (Stack (public), Stack (private)), Get-Set Idiom, and Summary. Below the menu, there is a small circular profile picture of a person. The main content area contains two bullet points:

- Understand access specifiers in C++ classes to control the visibility of members
- Learn to design with Information Hiding

At the bottom of the slide, there is footer text: NPTEL MOOCs Programming in C++, Partha Pratim Das, and the number 2.

So, our objectives are to understand access specifiers and to learn to design with information hiding.

(Refer Slide Time: 02:15)



The slide has a blue header bar with the title "Module Outline". On the left, there is a vertical sidebar with a navigation menu. The menu items are identical to the previous slide: Module 12, Partha Pratim Das, Objectives & Outline, Access Specifiers (public and private), Information Hiding (Stack (public), Stack (private)), Get-Set Idiom, and Summary. Below the menu, there is a small circular profile picture of a person. The main content area contains three bullet points:

- Access specifiers
 - public Access Specifier
 - private Access Specifier
- Information Hiding
 - Stack with public data
 - Stack with private data
- Get-Set Idiom

At the bottom of the slide, there is footer text: NPTEL MOOCs Programming in C++, Partha Pratim Das, and a small circular video thumbnail of the speaker.

There are two kinds of access specifier, and we will show how they can be used to hide information and we will end with a specific Get-Set idiom.

(Refer Slide Time: 02:26)

Module 12	Program 12.01/02: Complex Number: Access Specification
Partha Pratim Das Objectives & Outline Access Specifiers public and private Information Hiding Block (public) Block (private) Get-Set Idioms Summary	<p>Public data, Public method</p> <pre>#include <iostream> #include <cmath> using namespace std; class Complex { public: double re, im; public: double norm() { return sqrt(re*re + im*im); } void print(const Complex t) { // Global fn. cout << t.re << "j" << t.im << endl; } int main() { Complex c = { 4.2, 6.3 }; // Okay print(c); cout << c.norm(); return 0; }</pre> <ul style="list-style-type: none">▪ public data can be accessed by any function▪ norm (method) can access (re, im)▪ print (global) can access (re, im)▪ main (global) can access (re, im) & initialize <p>Private data, Public method</p> <pre>#include <iostream> #include <cmath> using namespace std; class Complex { private: double re, im; public: double norm() { return sqrt(re*re + im*im); } void print(const Complex t) { // Global fn. cout << t.re << "j" << t.im << endl; // 'Complex::re': cannot access private // member declared in class 'Complex' // 'Complex::im': cannot access private // member declared in class 'Complex' } int main() { Complex c = { 4.2, 6.3 }; // Error // 'initializing': cannot convert from // 'initializer-list' to 'Complex' print(c); cout << c.norm(); return 0; }</pre> <ul style="list-style-type: none">▪ private data can be accessed only▪ norm (method) can access (re, im)▪ print (global) cannot access (re, im)▪ main (global) cannot access (re, im)

Start with the complex number example. We have seen this example before and now we are not looking at the C versions of the code anymore. We are looking at C++ class only, so the left column and the right column both, show the C++ class complex; they have the same set of data members as we had already seen. But there is one difference if you see in the way they are specified. Here before the data members, there is a keyword public on the left-hand side and there is a keyword private on the right-hand side. These keywords are known as Access Specifier keywords and there are two of them for now to understand - a public specifier and a private specifier.

Earlier in module 11, every example, we saw was using public specifier. Which means that if you now look into the function that is written right after this the print function, which takes the instance of a complex object as a constant reference, it can print the components of that object; so, what it has to do for printing; given the object t, which it gets as a parameter, it has to refer to t.re. That it has to refer to the specific data member of the class and it is allowed to do so. This will come as a little surprised that I am saying that is allowed to do so, because if instead of writing complex class if you would have written struct complex in C you would have always done only this because that is the only way C does it.

So, there is no surprise on this part if I say my data member is public then any method any function that I have globally defined, I can once I have the object of this class I can always access the data members and look what is there inside. So, using that we can define initialize an object and we can print the object.

Now, it may also be noted that this class complex also has a method norm and that method is also specified to be public. So, which means that if I have in main if I have an object c I can invoke this method norm and I will be allowed to do that and I can either use a global function print to print the values or I can use a method norm() to compute the norm of that complex number.

Now, let us look at the right-hand side, here if you just compare the method norm is still called public, but the data members are now set to be private. Now look at the same norm function to print the complex number is a same code, but now if you try to compile it you find this kind of what is shown as commands or actually not part of the code.

If you just try to compile this with the C++ compiler these are the kind of compiler error messages that you will get. It says, for example, I will read out here complex::re cannot access private member declared in class complex. Which mean that if I have a global function as in here and have an object, which needs to take or access the private data member of the class, the private data member of the class then I am not allowed to do so. That is what is, but in this case, I was allowed to do so, here I am not allowed to do so. That is the difference between having a public access specifier or a private access specifier.

If you have a private access specifier then you are not allowed to access that member from outside the class, from a global function or some of the external functions. But if you look at the member function norm, member function norm is also trying to access the same component re; is trying to access the same component re, but the compiler does not complain about it, because the norm is a member function of class. So, it is considered that norm will have the same kind of access rights, visibility rights as anyone else in that class, but a global function like print which is outside the class cannot have

the private access, kind of, the private access is a privileged to the members of the class only.

So, if we summarize, just check the comments below. So, when the access of the data members is public, then data can be accessed by any function. When the access specifier of the data members is private, it can be accessed only by the methods of that class and any global function or the main function will not be able to access and manipulate and initialize those data members. This is what is known as a basic concept of access specification or it is also called feasibility restriction. Because as if something which is public, which you can sit outside the class in a global function or in the member function of some other class you can sit there and be able to see what is happening in all public members. But in terms of the private members you do not have the privilege of doing that.

(Refer Slide Time: 09:36)

The slide has a blue header with the title 'Access Specifiers'. On the left, there's a sidebar with a logo, the text 'Module 12 Partha Pratim Das', and a navigation menu with items like 'Objectives & Outcome', 'Access Specifiers public and private', 'Information Hiding Static (public) Static (private)', 'Get Set Miss', and 'Summary'. The main content area contains a bulleted list of points about access specifiers:

- Classes provide **access specifiers** for members (data as well as function) to enforce **data hiding** that separates **implementation** from **interface**
 - **private** - accessible inside the definition of the class
 - member functions of the same class
 - **public** - accessible everywhere
 - member functions of the same class
 - member function of a different class
 - global functions
- The keywords **public** and **private** are the *Access Specifiers*
- Unless specified, the access of the members of a class is considered **private**
- A class may have multiple access specifier. The effect of one continues till the next is encountered

At the bottom, there are footer links for 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das', and a page number '5'.

So, access specifiers, classes provide access specifiers for members and the access specifiers are available for data as well as for function we have shown example for data only for private, but they are available for both and they are used to enforced data hiding. So, just to formally put, if some member is private it is accessible inside the definition of the class or which means that it is accessible from the member functions, accessible from

the member functions of the class alone and not from any other many member function for some other class or from global functions. But if it is a public access then that member is accessible from naturally the member functions of the same class member functions of other classes, global functions and so on.

Public and private are the two keywords for access specification. By default, if you have not, if you have just set class complex or class rectangle and we have not specified any access, then by default the access of the members of a class is considered to be private. And otherwise, we have to write private: or not public: to say what the access specification is and I can write it multiple times, every time I write access specification, that specification will continue to apply till a next specification is met. It could be again a public specification could be followed by another public specification or by a private specification and so on and as many of them as you want can be put.

(Refer Slide Time: 11:29)

The slide has a blue header bar with the title 'Information Hiding'. On the left, there is a vertical sidebar with a logo at the top, followed by a list of navigation links: 'Module 12', 'Partha Pratim Das', 'Objectives & Outcome', 'Access Specifiers (public, protected, private)', 'Information Hiding (Implementation, Interface, Encapsulation)', 'Get Set Done', and 'Summary'. The main content area contains a bulleted list:

- The private part of a class (*attributes and methods*) forms its **implementation** because the class alone should be concerned with it and have the right to change it
- The public part of a class (*attributes and methods*) constitutes its **interface** which is available to all others for using the class
- Customarily, we put all *attributes* in private part and the *methods* in public part. This ensures:
 - The **state** of an object can be changed only through one of its *methods* (with the knowledge of the *class*)
 - The **behavior** of an object is accessible to others through the *methods*
- This is known as **Information Hiding**

At the bottom of the slide, there is footer text: 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

Now, with this very tiny concept of access restriction or visibility restriction, we create what is known as the major infrastructure or major paradigm of object-oriented thinking object-oriented programming, which is known as information hiding. To be say, the private part of a class, private part of a class that is the attributes and method forms its implementation. Because if it is private then the class alone can change it, class alone has

a right to change it and if it is an implementation the class alone should be concerned with it, others should not be concerned with what is there in the implementation.

Whereas the public part of a class attributes and methods both constitutes in interface which others can see, others can access, is available to anyone, who may want to use any instance of this class. So, private very strongly relates to implementation, public strongly relates to interface.

Now it is customarily, we will see this customarily, but this is a very strong design guideline that will consistently follow, customarily we will put all attributes that is data members in private please try to remember this and we put all this methods in public. So, if we put all data members in private what does it mean? What does the combination of values in the data members decide the combination of values in the data member we have seen in the last module decide the state of the object. So, the state of the object remains private, is a part of the implementation and it can be changed only through one of its methods.

Because, all data members are private if all attributes are private they cannot be changed additct the only way you can change them is to use some member function we change them, which means that the object will always get to know that you have changed them because a function has been called. Whereas the behavior of the object, which is basically the collection of the methods that it supports, is accessible through other methods and it is accessible to others because that is the way the behavior is what you are offering to the external world.

So, if we talk little bit concrete in terms of example, that we have already seen, we have seen the example of a stack having an array of characters and a top marker. This character array in the top marker should have a visibility of private, that is they define the state of the object, state of the stack. And nobody should be able to change the top marker or write something arbitrarily in any location of the array, because the stack needs to maintain the last in first out property.

Whereas, the behavior of the stack will be defined by the methods that it supports, specifically the empty, top, push and pop, which together allows us to define the LIFO behavior of the whole data structure. And this whole paradigm of separating out implementation from the interface separating or protecting the state behind the private visibility and exposing the behavior with the public visibility is known as information hiding in object-oriented programming, alternate names follows like encapsulation, state base design and so on. But information hiding and encapsulation are the common names that we will try to follow here.

(Refer Slide Time: 15:44)

The slide has a blue header with the title 'Information Hiding'. On the left, there's a sidebar with navigation links: 'Module 12', 'Partha Pratim Das', 'Objectives & Outline', 'Access Specifiers (public and private)', 'Information Hiding', 'Stack (public)', 'Stack (private)', 'Get-Set Mism.', and 'Summary'. The main content area contains the following text and bullet points:

For the sake of efficiency in design, we at times, put *attributes* in **public** and / or *methods* in **private**. In such cases:

- The **public attributes** *should not* decide the *state* of an object, and
- The **private methods** cannot be part of the behavior of an object

We illustrate information hiding through two implementations a stack

NPTEL MOOCs Programming in C++ Partha Pratim Das 7

Now obviously, when we learnt programming, when we learnt software design; a design can never be a something, which is very strongly cast in concrete. So, there are always some scopes for doing exceptions. And so, for the sake of the efficiency we will show later on, that for the sake of efficiency at times we put some attributes in public, which goes against as we understand by now against the principle of information hiding or we can put some methods in private, which again goes against information hiding; because a method which is in private is not a part of the interface it has to be a part of the implementation.

So, the public attributes should be such that they do not decide, primarily the state of the object and the private methods cannot be the part of the behavior. So, just keep this in mind follow the rule principle of information hiding that is attributes are private and methods are public for the behavior and the state, but we will come across such exceptions, where we will show why this kind of changes are required in some designs.

Now, let us pick up two implementations of a stack to specifically discuss and illustrate this principle of information hiding.

(Refer Slide Time: 17:14)

Program 12.03/04: Stack: Implementations using public data

Module 12

Partha Pratim Das

Objectives & Outline

Access Specifiers

public and private

Information Hiding

Stack (public)

Stack (private)

Get-Sets Missing

Summary

Using dynamic array	Using vector
<pre>#include <iostream> #include <cstdlib> using namespace std; class Stack { public: char *data_; int top_;</pre>	<pre>#include <iostream> #include <vector> using namespace std; class Stack { public: vector<char> data_; int top_;</pre>
<pre>public: int empty() { return (top_ == -1); } void push(char x) { data_[top_+1] = x; } void pop() { --top_; } char top() { return data_[top_]; }</pre>	<pre>public: int empty() { return (top_ == -1); } void push(char x) { data_.push_back(x); } void pop() { --top_; } char top() { return data_.back(); }</pre>
<pre>};</pre>	<pre>};</pre>
<pre>int main() { Stack s; char str[10] = "ABCDE"; s.data_ = new char[100]; // Exposed Init s.top_ = -1; // Exposed Init for(int i = 0; i < 5; ++i) s.push(str[i]); while (!s.empty()) { cout << s.top(); s.pop(); } // Outputs: EDCBA -- Reversed string delete [] s.data_; // Exposed De-Init return 0; }</pre>	<pre>int main() { Stack s; char str[10] = "ABCDE"; s.data_.reserve(100); // Exposed Init s.top_ = -1; // Exposed Init for(int i = 0; i < 5; ++i) s.push(str[i]); while (!s.empty()) { cout << s.top(); s.pop(); } // Outputs: EDCBA -- Reversed string return 0; }</pre>

- public data reveals the internals of the stack (no information hiding)
- Spills data structure codes (Exposed Init / De-Init) into the application (main)
- To switch from array to vector or vice-versa the application needs to change

NPTEL MOOCs Programming in C++
Partha Pratim Das

Here is a stack; here you are already very well familiar with the notion of the stack and how it is written. So, here you can see that we have the data of the stack as public access here, also we have the data of stack as public access the members methods are all public access. So, it is not strictly following the principle of information hiding it has exposed the internal data. So, since it has exposed the internal data and why does it need to expose the internal data, because if I have the stack defined like this I certainly need to do two things on it I need to allocate an appropriate sized array in the data_ member of the stack class, so that the elements can be put in that array and I also need to initialize the top index to -1.

So, these are points where the internal state of the object is exposed and that is using the public access specification we can actually make use of that and then given that we can use the reversed string function, realize the reversed string functionality by making use of the different stack functions.

Similarly, here we show another example here the data type that we use as a container is a vector you will recall, we had discussed this in the early modules, but we talked about array vector duality and we showed how vectors can be very effective in doing this. But even if we use vectors then also we need to possibly do an initialization to give its initial size and of course, the index marker needs to be initialized the top marker needs to be initialized. So, if we use the public data that is if we do not enforce information hiding then you can very well see that between these two codes, which both of each are actually trying to take a string and reverse it.

There are a lot of statements that we need to put, which is basically trying to initialize the stack according to how it is implemented and so on. But the basic question is if I have a string and I want to use a stack of characters to reverse it why should I, be bothered about, whether the stack is dynamically allocated using a character array or it uses as a vector or automatically allocated static character array and so on. Why should I be at all concerned with it, why should the application at all need to know all these exposed initialization and de-initialization parts of the code? So, let us move on and see what other things can happen with the public data.

(Refer Slide Time: 20:22)

```

Module 12
Partha Pratim Das
Objectives &
Outline
Access
Specifiers
public and private
Information
Holding
Stack (public)
Stack (private)
Get-Set Method
Summary

```

Using dynamic array	Using vector
<pre> #include <iostream> #include <cstdlib> using namespace std; class Stack { public: char *data_; int top_; public: int empty() { return (top_ == -1); } void push(char x) { data_[top_+1] = x; } void pop() { --top_; } char top() { return data_[top_]; } }; int main() { Stack s; char str[10] = "ABCDE"; s.data_ = new char[100]; // Exposed Init s.top_ = -1; // Exposed Init for(int i=0; i<5; ++i) s.push(str[i]); s.top_ = 2; // STACK GETS INCONSISTENT while (!s.empty()) { cout << s.top(); s.pop(); } // Outputs: CBA -- WRONG!!! delete [] s.data_; // Exposed De-init return 0; } </pre>	<pre> #include <iostream> #include <vector> using namespace std; class Stack { public: vector<char> data_; int top_; public: int empty() { return (top_ == -1); } void push(char x) { data_.push_back(x); } void pop() { --top_; } char top() { return data_[top_]; } }; int main() { Stack s; char str[10] = "ABCDE"; s.data_.reserve(100); // Exposed Init s.top_ = -1; // Exposed Init for(int i=0; i<5; ++i) s.push(str[i]); s.top_ = 2; // STACK GETS INCONSISTENT while (!s.empty()) { cout << s.top(); s.pop(); } // Outputs: CBA -- WRONG!!! return 0; } </pre>

• Application may intentionally or inadvertently tamper the value of top_ - this corrupts the stack
 • s.top_ = 2; destroys consistency of the stack and causes wrong output

NPTEL MOOCs Programming in C++ Partha Pratim Das 9

The same example, but just think of just. I would like to just draw your attention to two lines one on the left here and one on the right here. Certainly, we will ask why should somebody do it what it is doing it is taking the value of a s.top and assigning two into that in both cases now. If you do that what it means your top marker as been corrupted your top marker has being changed, which means the stack has become inconsistent, if you just now carefully look in to this code this is a string containing 5 characters.

So, this for loop runs for 5 times you have done 5 pushes. So, the top marker has got incremented 5 times for standing from minus 1 is gone through 0, 1, 2, 3, 4 and now we are just forcing it to a value 2, which means that the top most two elements, which are actually existing in the stack logically does not exist with the stack any more.

So, if you now complete this code then it prints C B A instead of it this should have printed E D C B A. But what has happened in the stack after it had done all the inserts A, B, C, D, E; 1, 2, 3, 4; I certainly forced the s.top, top of the stack to two, which means that these two elements which are logically actual in the stack simply disappeared. So, the risk of exposing the data how could I do this, how could I change this, I do not know how the stack is managed this, but this was possible because I could access the top component of this class, the top data member of this class. So, exposing the

implementation using public data has severe risks and that risk is what we are trying to avoid in terms of designing such classes.

(Refer Slide Time: 22:43)

```

Module 12
Partha Pratim Das
Objectives & Outcome
Axiom Specifiers
push and pop
Information Hiding
Stack (public)
Stack (private)
Get Set Method
Summary

Program 12.05/06: Stack:
Implementations using private data – Safe

Using dynamic array


```

#include <iostream>
using namespace std;
class Stack {
private:
 char *data_; int top_;
public:
 // Initialization
 Stack() : data_(new char[100]), top_(-1) {}
 // De-Initialization
 ~Stack() { delete[] data_; }
 int empty() { return (top_ == -1); }
 void push(char x) { data_[++top_] = x; }
 void pop() { --top_; }
 char top() { return data_[top_]; }
};

int main() {
 Stack s; char str[10] = "ABCDE";
 for (int i=0; i<5; ++i) s.push(str[i]);
 while (!s.empty()) {
 cout << s.top(); s.pop();
 }
 return 0;
}

```



Using vector


```

#include <iostream>
#include <vector>
using namespace std;
class Stack {
private:
 vector<char> data_; int top_;
public:
 // Initialization
 Stack() : top_(-1) { data_.resize(100); }
 // De-Initialization
 ~Stack() {}
 int empty() { return (top_ == -1); }
 void push(char x) { data_[++top_] = x; }
 void pop() { --top_; }
 char top() { return data_[top_]; }
};

int main() {
 Stack s; char str[10] = "ABCDE";
 for (int i=0; i<5; ++i) s.push(str[i]);
 while (!s.empty()) {
 cout << s.top(); s.pop();
 }
 return 0;
}

```


```

• private data hides the internals of the stack (information hiding)
 • Data structure codes contained within itself with initialization and de-initialization
 • To switch from array to vector or vice-versa the application needs no change
 • Application cannot tamper stack – any direct access to top_ or data_ is compilation error!

NPTEL MOOCs Programming in C++ Partha Pratim Das 10

Now we present the same design exactly the same code with a very minor change that is we now have the data put in as private access. So, now, we are following the principle of information hiding. What is the consequence? One consequence is the whole of the stack whether you do it by this or you do it by this is completely internal to the stack class. Which mean that certainly nothing regarding them can be done from the application, because the application will have no right to access the data_ or top_ members.

So, which means that the code, the application code, the two main functions now get absolutely identical, you can just refer one slide backward, you can just refer one slide backward we were here or if we go another slide backward we are here. And if you now compare the two main functions you can see that all those Exposed Init, Exposed De-Init kind of lines are different between the two main functions. But both of them are trying to do the same string reverse, but they are just that they are using a stack whose internal container is different.

(Refer Slide Time: 24:15)

Module 12
Partha Pratim Das

Objectives & Outline
Access Specifiers
public and private
Information Hiding
Stack (public)
Stack (private)

Get-Set-Move
Summary

Program 12.05/06: Stack:
Implementations using private data – Safe

Using dynamic array

```
#include <iostream>
using namespace std;
class Stack { private:
    char *data_; int top_;
public:
    // Initialization
    Stack() : data_(new char[100]), top_(-1) {}
    // De-Initialization
    ~Stack() { delete[] data_; }
    int empty() { return (top_ == -1); }
    void push(char x) { data_[++top_] = x; }
    void pop() { --top_; }
    char top() { return data_[top_]; }
};

int main() {
    Stack s; char str[10] = "ABCDE";
    for (int i=0; i<6; ++i) s.push(str[i]);
    while (!s.empty()) {
        cout << s.top(); s.pop();
    }
    return 0;
}
```

Using vector

```
#include <iostream>
#include <vector>
using namespace std;
class Stack { private:
    vector<char> data_; int top_;
public:
    // Initialization
    Stack() : top_(-1) { data_.resize(100); }
    // De-Initialization
    ~Stack() {}
    int empty() { return (top_ == -1); }
    void push(char x) { data_.at(top_) = x; }
    void pop() { --top_; }
    char top() { return data_.at(top_); }
};

int main() {
    Stack s; char str[10] = "ABCDE";
    for (int i=0; i<6; ++i) s.push(str[i]);
    while (!s.empty()) {
        cout << s.top(); s.pop();
    }
    return 0;
}
```

* private data hides the internals of the stack (information hiding)
* Data structure codes contained within itself with initialization and de-initialization
* To switch from array to vector or vice-versa the application needs no change
* Application cannot tamper stack – any direct access to top_ or data_ is compilation error!

NPTEL MOOCs Programming in C++ Partha Pratim Das 10

But suppose now we have made these private and as we have made them private, then certainly we cannot access them and therefore, the application functions become identical no changes can be made. There is no risk of assigning 2 to s.top as we did last time because if we try to somewhere here we were doing it in the last example. If we try to now write s.top_ = 2 here, then simply I will have a compilation error, the my compiler will tell me that s is of stack type, where top is a private data member. So, from outside the class stack main is outside of the class stack it is a global function with respect to class stack, I am not allowed to access s.top and therefore, such risks will not exist.

So, certainly with this principle of information hiding I can have a much better design, a much better implementation, where the private data, the internals are completely hidden, data structure is contained all within itself. And if I want to switch from one implementation to the other, from left to right or from right to left, my application code will not need to make any change what so ever at all. So, that is the basic principle that we will try to follow in terms of this design.

(Refer Slide Time: 25:45)

The slide has a blue header bar with the title 'Interface and Implementation'. Below the header is a navigation menu on the left:

- Module 12
- Partha Pratim Das
- Objectives & Outline
- Access Specifiers (public, private, protected)
- Information Hiding
- Stack (public)
- Stack (private)
- Get-Set Method
- Summary

The main content area is divided into three columns:

Interface	Implementation	Application
<pre>// File: Stack.h class Stack { private: // Implementation char *data_; int top_; public: // Interface Stack(); ~Stack(); int empty(); void push(char x); void pop(); char top(); };</pre>	<pre>// File: Stack.h class Stack { private: // Implementation char *data_; int top_; public: // Interface Stack(); ~Stack(); int empty(); void push(char x); void pop(); char top(); }; // File: Stack.cpp // Implementation Stack::Stack() : data_(new char[100]), top_(-1) {} Stack::~Stack() { delete[] data_; } int Stack::empty() { return (top_ == -1); } void Stack::push(char x) { data_[++top_] = x; } void Stack::pop() { --top_; } char Stack::top() { return data_[top_]; }</pre>	<pre>#include "Stack.h" int main() { Stack s; char str[10] = "ABCDE"; for (int i = 0; i < 5; ++i) s.push(str[i]); while (!s.empty()) { cout << s.top(); s.pop(); } return 0; }</pre>

At the bottom of the slide, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

Now given this let me just briefly discuss, how should the basic interface and implementation be organized. So, here if we look at, this is stack class and we will expect that this comes in a header file; where header file is which defines, what the different classes are there in their members and so on. And if you look into this, this part is private, which is part of the implementation and this part is an interface, which everybody needs to know. So, if you looking to the implementation again you just have another copy of this and this will be required in the implementation as well as in the application below.

Implementation will need it, implementation means which ever will write the body of this functions the definition of this functions these codes, which certainly go in a separate .cpp file, that is certainly needs to know what the name of the class, it needs to know, what are different methods and so on.

So, this header has to go here, but to be able to use this header you do not need to know whether its data member is allocated dynamically or it is automatically allocated and so on. All that you need to know are that there are different data members like in here which can be used in the push pop empty all this kind of methods.

So, to summarize we will have one header file, which has the class that primarily should show the interface there should be one implementation file for the class that has the codes of all the methods and should include this. So, you can see this is what I am calling Stack.h, this is what I called is Stack.cpp. And then I separate application file whatever the name of the that application file is, which should include the same header file, make sure that it is the same class definition that is used between the implementation and the application this is where you write the application code.

So, this is the basic structure of any good object-oriented C++ design that we will try to follow, which perfectly allows us to perform information hiding. You will of course, raise some objection to the fact that the application does get to see that it cannot access the private data members, but that application still gets to see that what private data members I have. So is that knowledge necessary for the application. We will answer that much later when we have studied some bit more of C++ mechanisms, particularly you have studied about inheritance. We will be able to see that actually that two is not necessary and interface can truly just be the public methods that a class support. Everything that is private could be defined separately also.

But for now, let us just take it that implementation and interface will be separated out a combined in a header file that defines the class data members and the class methods as signature. A class implementation file, which has all the implementation codes of the methods, and the application can include the class header file will include the class header file use the methods and would be able to work out on the application.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 22
Access Specifiers (Contd.)

Welcome to part 2 of module 12 of Programming in C++. We have discussed in this module the different access specifier, so the visibility options of member of a class. We have learnt that there are public and private access specifiers which can be used to control who can access which part of the class. Public specifiers means that anyone, that is any global function or any other class can access the member which is specified as public, but if some member data member or method is specified as private then that data member or method can be accessed only from within the definition of the class or more precisely from other methods of the same class, but it cannot be accessed by anyone else.

Based on this we have outlined the principle of information hiding where we have prescribed that we should always have the attributes or data members as private and the methods as public giving us the basic hiding of the state from the behavior of the object. And we have seen example of that using a stack showing that if you expose the implementation if you put the data members in public then what kind of difficulties and risk that one would run, instead if the data members are private and only the methods of the stack type or stack class, push, pop, top and empty are public then an application would be able to seamlessly use a stack without being concerned about how the stack is actually implemented without running any risk of tampering the stack in the process of writing the application. Finally, we have shown how typically a code object-oriented C++ code should be organized for hiding information in terms of the header and implementation files.

(Refer Slide Time: 02:41)

**Get-Set Methods:
Idiom for fine-grained Access Control**

Module 12
Partha Pratim Das

Objectives & Outline
Access Specifiers (public and private)
Information Hiding
Stack (public) Stack (private)

Get-Set Idiom
Summary

- As noted, we put all *attributes* in **private** and the *methods* in **public**. This restricts the access to data completely
- To fine-grain the access to data we provide selective public member functions to *read* (get) and / or *write* (set) data

```
class MyClass { // private
    int readWrite_; // Like re_, im_ in Complex -- common aggregated numbers
    int readOnly_; // Like DateOfBirth, Emp_ID, RollNo -- should not need a change
    int writeOnly_; // Like Password -- reset if forgotten
    int invisible_; // Like top_, data_ in Stack -- keeps internal state
public:
    // get and set methods both to read as well as write readWrite_member
    int getReadWrite() { return readWrite_; }
    void setReadWrite(int v) { readWrite_ = v; }

    // Only get method to read readOnly_member - no way to write it
    int getReadOnly() { return readOnly_; }

    // Only set method to write writeOnly_member - no way to read it
    void setWriteOnly(int v) { writeOnly_ = v; }

    // No method accessing invisible_member directly - no way to read or write it
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 12

We will next talk about an Idiom, which is commonly known as the Get-Set Idiom which is used for fine grained access control in different members of a class or an object. We know that we will put all attributes in private and we will put methods in public and this restricts access to data completely which is, what one extreme that you can have. Now typically we would like to fine grain this access and we would like to provide selective public member functions or read functions or get methods or write functions or set methods on the data that we have protected by making them private.

So, I just illustrate the whole idea in terms of an example, here is a class which I just call it as a class MyClass which is not very important. There are four data members assumed and I have not used any access specifier after the beginning of the class definition here which means that by default these data members are private there in, and these data members are intended in a way that the first data members read write is a data member which, as I being a data member it is private so nobody can directly come and change its value, but in terms of its use I would like to read and write its value. For example, this is a member like a re components of a complex number, im component of a complex number or so, I would need to read and set the value of this component if I am reading a complex number. If I am writing the current value of a complex number I will need to read the re, im component and then so on. A large number of data member aggregate

members turned out to be read write. So, I need to provide some mechanism by which I should be able to do that and the simple way to do this is to provide a pair of get-set functions.

What is the get, set function do? A get function say on the read write on this variable say has a name `getReadWrite`, it is simply takes this data member and returns it. This method is kept in the public space, so any external function or any other member functions of any other class can call the `getReadWrite` functions seamlessly, because it is available in the public space. And this in turn will access the private member and return its value. So in about a way we are selectively making this the value of this known to the external world with the control that whenever somebody has to do that somebody has to go through this particular (Refer Time: 05:46). For example, one what is the basic difference between providing this get and let us say similarly I have a set which can take a value `v` and actually a science is valued to the data members read write. So basically, if I have get and set both then I can read as well as write the value of this variable `readWrite`.

Now you will wonder an (Refer Time: 06:14) question has to why do you want to make this private and then put a pair of public methods which can get and set this, we could have just made this public and anybody could have change that. But yes, in terms of read write this is similar, but there is a major difference because if I put it in public then the object will never get to know, when it is being this particular data member is being read or when it is being read. But if it is done through a function is a method then when it is being read I can also do some computation here. When is being written I can also do some computation before or after this value is written it to. So, the object will always be aware that the value of read write has been changed or the value of read write has been read by somebody. So, it is not exactly seem as to putting this data member as a public access.

Now, coming to the other data members if we look into the next one we want to just make it read only. So, there is a lot of in terms of any model different real-world data types, encapsulations we will find that there are lot of data which is read only like date of birth. Date of birth of a person cannot keep on changing so it should be set to something when I create the object for that person and after that it should not be possible to change,

is the best way to control that would be to make this a with the private access and then allow just a get function on it. So, which will allow somebody to read the date of birth and possibly compute age of that person or read the employee Id and check out, read the roll number of the student, and find the grade of the student and so on. All these get function will allow us to get that data but will not be able to change that data. So, which means that this particular if I just provide the get function and the responding data becomes read only data.

Once you get this then obviously rest becomes very straight forward it is just completing the all possible combinations I can have a right only variable if you just have a set method on that variable but there is no get method do not wonder variables can be write only there are several variables which can be write only. One very common example is password, because password is so sensitive that normally you would not like to allow any mechanism at all to read a password all that you might want, might allow will have to allow is the is a mechanism to change the password when needed. So, password is something which you just write you never read that values. You can just have only set and not no get functions which makes it a write only.

Similarly, in the final group you may have invisible members like the top of the stack or the data array of the stack you just do not want to know how the stack operates all that you want to know is push, pop, top and empty you should work. So, there should be no method at all either to read the value of top or data or to change/set the value of top or data, and therefore there should be no set-get methods on that. So, with this set-get idiom as you can see that we can create a very fine grain control on the accessibility of the data members and similar accessibility of different methods can also be done, but certainly get-set is a is a typical idiom that works for data members to allow you to do more finite things.

(Refer Slide Time: 10:05)

The slide has a blue header bar with the title "Module Summary". On the left, there is a sidebar with a logo at the top, followed by a vertical list of topics: "Module 12", "Partha Pratim Das", "Objectives & Outline", "Access Specifiers (public and private)", "Information Hiding", "Block (public) Block (private)", "Get-Set idiom", and "Summary". The main content area contains a bulleted list of three points:

- Access Specifiers helps to control visibility of data members and methods of a class
- The private access specifier can be used to hide information about the implementation details of the data members and methods
- Get, Set methods are defined to provide an interface to use and access the data members

At the bottom of the slide, there is footer text: "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and "13".

With this we will close on this module. In this module we have understood access specifiers how they help to control visibility of the class members, the public, and the private. They can be used as we have seen to hide information about the implementation of a class while exposing the behavior through the interface and we have specifically seen that get-set methods a specific idiom is often used in C++ to provide very fine grain control on the interface in terms of what access to the data members you want to provide in your design.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 23
Constructors, Destructors and Object Lifetime

Welcome to module 13 of Programming in C++.

(Refer Slide Time: 00:30)



Module Objectives

- Understand Object Construction (Initialization)
- Understand Object Destruction (De-Initialization)
- Understand Object Lifetime

NPTEL MOOCs Programming in C++ Partha Pratim Das 2

In this module, we will talk about understanding how objects are constructed or initialized, and after their use how they are destructed or de-initialized, and in this whole process what is the life time that the object will have. In terms of the object-oriented futures in C++ we have already seen how classes can be defined and how objects of a certain class can be instantiated.

We have already had a look in the data members and methods and we have talked about the whole paradigm of information hiding, the basic design process of access specification and visibility restriction to make data members particularly private in a design and provide an interface to methods which is public. Now, we will extend on that and talk more, this was more on the design aspect in terms of the structure of a class or

the blue print of different objects. Now in the current module will look specifically on the runtime execution, runtime behavior of an object starting trying to find out when does it get into life, and how long does it remain in life? And what happens at the end of life of an object?

(Refer Slide Time: 02:07)

The slide has a dark blue header with the title "Module Outline". On the left, there is a vertical sidebar with a logo at the top, followed by a list of topics: "Module 13", "Partha Pratim Das", "Objectives & Outline", "Constructor Parameterized Overloaded", "Destructor", "Default Constructor", "Object Lifetime", "Automatic Static Dynamic", and "Summary". The main content area contains a bulleted list of topics: "Constructors" (Parameterized, Overloaded), "Destructor", "Default Constructor", and "Object Lifetime" (Array, Dynamic). At the bottom, there is footer text: "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and the number "3".

We will get started with all these discussions, this is outline does it can see on the left of your panel also.

(Refer Slide Time: 02:13)

Program 13.01/02: Stack: Initialization					
Module 13 Partha Pratim Das Objectives & Outline Constructor Parameterized Overloaded Constructor Default Constructor Object Lifetime Automatic Static Dynamic Summary	<table border="1" style="width: 100%;"> <thead> <tr> <th style="text-align: center;">Public Data</th> <th style="text-align: center;">Private Data</th> </tr> </thead> <tbody> <tr> <td style="padding: 10px;"> <pre>#include <iostream> using namespace std; class Stack { public: char data_[10]; int top_;</pre> <pre>public:</pre> <pre> int empty() { return (top_ == -1); } void push(char x) { data_[++top_] = x; } void pop() { --top_; } char top() { return data_[top_]; }</pre> <pre>}; int main() { char str[10] = "ABCDE"; Stack s; s.top_ = -1; // Exposed initialization for (int i = 0; i < 6; ++i) s.push(str[i]); // s.top_ = 2; // RIBK while (!s.empty()) { cout << s.top(); s.pop(); } return 0; }</pre> <ul style="list-style-type: none"> • Spills data structure codes into application • public data reveals the internals • To switch container, application needs to change • Application may corrupt the stack! </td> <td style="padding: 10px;"> <pre>#include <iostream> using namespace std; class Stack { private: char data_[10]; int top_;</pre> <pre>public:</pre> <pre> void init() { top_ = -1; } int empty() { return (top_ == -1); } void push(char x) { data_[++top_] = x; } void pop() { --top_; } char top() { return data_[top_]; }</pre> <pre>}; int main() { char str[10] = "ABCDE"; Stack s; s.init(); // Clean initialization for (int i = 0; i < 6; ++i) s.push(str[i]); // s.top_ = 2; // Compile error - SAFE while (!s.empty()) { cout << s.top(); s.pop(); } return 0; }</pre> <ul style="list-style-type: none"> • No code in application, but init() to be called • private data protects the internals • Switching container is seamless • Application cannot corrupt the stack! </td> </tr> </tbody> </table>	Public Data	Private Data	<pre>#include <iostream> using namespace std; class Stack { public: char data_[10]; int top_;</pre> <pre>public:</pre> <pre> int empty() { return (top_ == -1); } void push(char x) { data_[++top_] = x; } void pop() { --top_; } char top() { return data_[top_]; }</pre> <pre>}; int main() { char str[10] = "ABCDE"; Stack s; s.top_ = -1; // Exposed initialization for (int i = 0; i < 6; ++i) s.push(str[i]); // s.top_ = 2; // RIBK while (!s.empty()) { cout << s.top(); s.pop(); } return 0; }</pre> <ul style="list-style-type: none"> • Spills data structure codes into application • public data reveals the internals • To switch container, application needs to change • Application may corrupt the stack! 	<pre>#include <iostream> using namespace std; class Stack { private: char data_[10]; int top_;</pre> <pre>public:</pre> <pre> void init() { top_ = -1; } int empty() { return (top_ == -1); } void push(char x) { data_[++top_] = x; } void pop() { --top_; } char top() { return data_[top_]; }</pre> <pre>}; int main() { char str[10] = "ABCDE"; Stack s; s.init(); // Clean initialization for (int i = 0; i < 6; ++i) s.push(str[i]); // s.top_ = 2; // Compile error - SAFE while (!s.empty()) { cout << s.top(); s.pop(); } return 0; }</pre> <ul style="list-style-type: none"> • No code in application, but init() to be called • private data protects the internals • Switching container is seamless • Application cannot corrupt the stack!
Public Data	Private Data				
<pre>#include <iostream> using namespace std; class Stack { public: char data_[10]; int top_;</pre> <pre>public:</pre> <pre> int empty() { return (top_ == -1); } void push(char x) { data_[++top_] = x; } void pop() { --top_; } char top() { return data_[top_]; }</pre> <pre>}; int main() { char str[10] = "ABCDE"; Stack s; s.top_ = -1; // Exposed initialization for (int i = 0; i < 6; ++i) s.push(str[i]); // s.top_ = 2; // RIBK while (!s.empty()) { cout << s.top(); s.pop(); } return 0; }</pre> <ul style="list-style-type: none"> • Spills data structure codes into application • public data reveals the internals • To switch container, application needs to change • Application may corrupt the stack! 	<pre>#include <iostream> using namespace std; class Stack { private: char data_[10]; int top_;</pre> <pre>public:</pre> <pre> void init() { top_ = -1; } int empty() { return (top_ == -1); } void push(char x) { data_[++top_] = x; } void pop() { --top_; } char top() { return data_[top_]; }</pre> <pre>}; int main() { char str[10] = "ABCDE"; Stack s; s.init(); // Clean initialization for (int i = 0; i < 6; ++i) s.push(str[i]); // s.top_ = 2; // Compile error - SAFE while (!s.empty()) { cout << s.top(); s.pop(); } return 0; }</pre> <ul style="list-style-type: none"> • No code in application, but init() to be called • private data protects the internals • Switching container is seamless • Application cannot corrupt the stack! 				
<small>NPTEL MOOCs Programming in C++</small>	<small>Partha Pratim Das</small>				
	<small>4</small>				

So, let us refer back to one of the earlier examples of stack we have already introduced this earlier. This stack has two data members, an array of characters to store the stack elements, a top marker which will keep the index of the top element and in the public it has the four LIFO methods, LIFO interface, empty, push, pop and top to use this stack and here we show the use of this stack to reverse a string.

Now, if we look into this then in a closer scrutiny you will find that here we are using this instantiating this stack to create a stack object, but as soon as we create this stack object we cannot start using it for doing the reverse string operation that we are having to do here, in between we need to make sure that the stack has a proper initialization, in the sense that once s is instantiated I need to know, ascertain, what is the value of the top marker at this point.

This is a point where no operation has happened so far, so you need to make sure that the stack just created conceptually has to be a null stack containing nothing an empty stack and therefore, its top marker must be one less than the first element that can go into the array that is the top marker must be -1. So, for this code to work it is critical that we add the initial value of top marker through an assignment after the stack variable as been instantiated.

If you look closely further you will realize that we are able to do this because I have defined the stack variables, the data members as public. Therefore, I can easily access the top_ data member and do this assignment and so therefore what we observed here is in this solution we have an initialization which exposes the internal of the stack in contrary to the information hiding principles that we had invited earlier.

So, let us move on to the right column where in contrast to the public data that we are used here, in contrast we use private data we go back to our information hiding paradigm so we make the data private and if we do that naturally, we cannot write this anymore here because as you can very well understand that if we try to write s.top_ = -1; then the compiler will not compile this program it will give us an error saying that top is a private data member and cannot be accessed. We solve that problem by introducing a separate function say by the name of init() which we put on the interface as a public method and init() basically initializes top to -1, and here instead of doing s.top = -1 we give a call to the init(). They should serve the same purpose.

With this we can still maintain the information hiding principle that the object-oriented design must follow and ensure that the solution will work. And a side benefit of doing this is also you can compare here in these two lines, for example when the data member was public as we had illustrated earlier in terms of access specification module that if the data member is public it is possible to create some potential risk, if inadvertently s.top is assigned to some value in the middle. But, when we are back to the information hiding principle now this kind of a code cannot be returned with a private data because it will become compilation error. In that way making the data member private and providing a init function is a better proposition of solution for the initialization problem, and certainly we would like to work on with this private data further.

(Refer Slide Time: 07:10)

Program 13.02/03: Stack: Initialization			
Module 13  Partha Pratim Das Objectives & Outline Constructor Parameterized Overloaded Default Constructor Object Lifetime Automatic Static Dynamic Sessions	<table border="1" style="width: 100%;"> <tr> <td style="padding: 10px; vertical-align: top;"> Using init() <pre>#include <iostream> using namespace std; class Stack { private: char data_[10]; int top_;</pre> <pre>public: void init() { top_ = -1; } int empty() { return (top_ == -1); } void push(char x) { data_[++top_] = x; } void pop() { --top_; } char top() { return data_[top_]; } };</pre> <pre>int main() { char str[10] = "ABCDE"; Stack s; s.init(); // Clean initialization for (int i = 0; i < 5; ++i) s.push(str[i]); // s.top_ = 5; // Compile error - SANE while (!s.empty()) { cout << s.top(); s.pop(); } return 0; }</pre> <ul style="list-style-type: none"> • <code>init()</code> serves no visible purpose – application may forget to call • If application misses to call <code>init()</code>, we have a corrupt stack </td> <td style="padding: 10px; vertical-align: top;"> Using Constructor <pre>#include <iostream> using namespace std; class Stack { private: char data_[10]; int top_;</pre> <pre>public: Stack() : top_(-1) {} // Initialization int empty() { return (top_ == -1); } void push(char x) { data_[++top_] = x; } void pop() { --top_; } char top() { return data_[top_]; } };</pre> <pre>int main() { char str[10] = "ABCDE"; Stack s; // Init by Stack::Stack() call</pre> <pre>for (int i = 0; i < 5; ++i) s.push(str[i]); while (!s.empty()) { cout << s.top(); s.pop(); } return 0; }</pre> <ul style="list-style-type: none"> • Can initialization be made a part of instantiation? • Yes. Constructor is implicitly called at instantiation as set by the compiler </td> </tr> </table>	Using init() <pre>#include <iostream> using namespace std; class Stack { private: char data_[10]; int top_;</pre> <pre>public: void init() { top_ = -1; } int empty() { return (top_ == -1); } void push(char x) { data_[++top_] = x; } void pop() { --top_; } char top() { return data_[top_]; } };</pre> <pre>int main() { char str[10] = "ABCDE"; Stack s; s.init(); // Clean initialization for (int i = 0; i < 5; ++i) s.push(str[i]); // s.top_ = 5; // Compile error - SANE while (!s.empty()) { cout << s.top(); s.pop(); } return 0; }</pre> <ul style="list-style-type: none"> • <code>init()</code> serves no visible purpose – application may forget to call • If application misses to call <code>init()</code>, we have a corrupt stack 	Using Constructor <pre>#include <iostream> using namespace std; class Stack { private: char data_[10]; int top_;</pre> <pre>public: Stack() : top_(-1) {} // Initialization int empty() { return (top_ == -1); } void push(char x) { data_[++top_] = x; } void pop() { --top_; } char top() { return data_[top_]; } };</pre> <pre>int main() { char str[10] = "ABCDE"; Stack s; // Init by Stack::Stack() call</pre> <pre>for (int i = 0; i < 5; ++i) s.push(str[i]); while (!s.empty()) { cout << s.top(); s.pop(); } return 0; }</pre> <ul style="list-style-type: none"> • Can initialization be made a part of instantiation? • Yes. Constructor is implicitly called at instantiation as set by the compiler
Using init() <pre>#include <iostream> using namespace std; class Stack { private: char data_[10]; int top_;</pre> <pre>public: void init() { top_ = -1; } int empty() { return (top_ == -1); } void push(char x) { data_[++top_] = x; } void pop() { --top_; } char top() { return data_[top_]; } };</pre> <pre>int main() { char str[10] = "ABCDE"; Stack s; s.init(); // Clean initialization for (int i = 0; i < 5; ++i) s.push(str[i]); // s.top_ = 5; // Compile error - SANE while (!s.empty()) { cout << s.top(); s.pop(); } return 0; }</pre> <ul style="list-style-type: none"> • <code>init()</code> serves no visible purpose – application may forget to call • If application misses to call <code>init()</code>, we have a corrupt stack 	Using Constructor <pre>#include <iostream> using namespace std; class Stack { private: char data_[10]; int top_;</pre> <pre>public: Stack() : top_(-1) {} // Initialization int empty() { return (top_ == -1); } void push(char x) { data_[++top_] = x; } void pop() { --top_; } char top() { return data_[top_]; } };</pre> <pre>int main() { char str[10] = "ABCDE"; Stack s; // Init by Stack::Stack() call</pre> <pre>for (int i = 0; i < 5; ++i) s.push(str[i]); while (!s.empty()) { cout << s.top(); s.pop(); } return 0; }</pre> <ul style="list-style-type: none"> • Can initialization be made a part of instantiation? • Yes. Constructor is implicitly called at instantiation as set by the compiler 		
<small>NFTEL MOOCs Programming in C++</small>	<small>Partha Pratim Das</small>		
<small>5</small>			

Now let us move on. On the left column you see the same code that you just saw which has the private data and public methods including and `init()` method to initialize `top`. As we have observed that provides a clean initialization, but the question is certainly what if the application forgets to call `init()`, or what if the application would call `init()` at a wrong position? If the application misses to call this `init()` which for the application as actually no purpose because the application is trying to reverse a string and it knows that it is a LIFO structure which has `empty`, `top`, `pop` and `push`, these four methods but, for all this to work it as to call in it but if it misses on that then naturally we have an arbitrary value in `top` to start with, and therefore the whole program will give a very unpredictable behavior.

So, you ask a question that can we do better than this, can we when the stack is actually instantiated when it is defined is it possible that right at that instantiation point if this initialization call can be somehow done. The mechanism of constructor basically provides this solution. C++ provides a mechanism by which if you instantiate an object a special function call the constructor is invoked right at this point, and this constructor can be used to initialize the values of data members as are required. Just looking into this example, then we do not have the `init()` call anymore this is not there instead we have introduced a new method which is called the constructor of this object. How do you

know this is the constructor? It has a typical signature of having the same name as the name of the class, so by the name you know this is a constructor it has a little bit of different way of putting things and we will discuss that more.

But, what we say here is necessarily we are saying that take the data member top and put -1 as an initial value. The other, it is not required to initialize the array because it will get used as and when push and pop happens. But the advantage is, if we define a constructor then as soon as the control will pass this particular point that is when the object gets instantiated immediately an automatic call, an implicit call, will happen to this method, and that call will make sure that top is indeed assigned to -1. So, when we return from this call at this point we already have the top of yes initialized to -1 and that, this implicit initialization is a beauty of the constructor.

(Refer Slide Time: 10:51)

```

Module 13
Partha Pratim Das
Objectives & Outcome
Constructor (Parameterized, Overloaded)
Destructor
Default Constructor
Object Lifetime
Automatic, Static, Dynamic, Functions

```

Automatic Array	Dynamic Array
<pre>#include <iostream> using namespace std; class Stack { private: char data_[10]; int top_= -1; // Automatic public: Stack(); // Constructor // Stack methods }; Stack::Stack() { top_ = -1; // Init List cout << "Stack::Stack() called" << endl; } int main() { char str[10] = "ABCDE"; Stack s; // Init by Stack::Stack() call for (int i=0; i<6; ++i) s.push(str[i]); while (!s.empty()) { cout << s.top(); s.pop(); } return 0; } Stack::Stack() called EDCB </pre>	<pre>#include <iostream> using namespace std; class Stack { private: char *data_; int top_= -1; // Dynamic public: Stack(); // Constructor // Stack methods }; Stack::Stack() : data_(new char[10]), top_(-1) { // Init cout << "Stack::Stack() called" << endl; } int main() { char str[10] = "ABCDE"; Stack s; // Init by Stack::Stack() call for (int i=0; i<6; ++i) s.push(str[i]); while (!s.empty()) { cout << s.top(); s.pop(); } return 0; } Stack::Stack() called EDCB </pre>
<ul style="list-style-type: none"> • top_ initialized to -1 in list • data_[10] initialized to new char[10] in list • Stack::Stack() called automatically when control passes Stack s; -- Guarantees initialization 	<ul style="list-style-type: none"> • top_ initialized to -1 in list • data_ initialized to new char[10] in list • Stack::Stack() called automatically when control passes Stack s; -- Guarantees initialization

NPTEL MOOCs Programming in C++ Partha Pratim Das 6

Let us look at more examples say here, let us now we again have the same kind of example with the stack where the constructor is there and we know on the left-hand side that the constructor will initialize top all that I am showing here is earlier the constructor was written as a part of the class right at this point you will remember. But now, we have written it outside of the class, it is written as Stack::Stack();, this naming also you will

recall is any class member has a name which is qualified by the name of the class. So, this is the name of the class Stack, this is the name of constructor.

What we have is the body of the constructor, and in that body first we have initialized top to -1 and then rest of the cout statement. If we do this then as soon as the stack is instantiated here the call is made to this point top is initialized the cout happens which you see here control comes back and then the reversal of the string happens. A very seamless procedure by which the top is getting explicitly initialized, of course the other data member which is a array is a automatic default initialization because it is a given array of certain size.

Now, let us see if we just make a small change instead of having an automatic array if you just have a pointer to character, therefore what we are trying to do is to dynamically create this array. So naturally in the stack code the initialization to top remains same, but we now also have to initialize the array pointer which we do by dynamically allocating it. At this point let us understand this way of writing the initialization. Just note carefully that in the initialization we first write the name of the data member and then within a pair of parenthesis we write the value that we want to use it as an initialization. If we look at data, this is the data pointer and this expression as you know will use the operator new allocate an array of character having 10 elements and return a pointer to that array having char * as a pointer type and that pointer is what is being set as a initial value in data.

This is something which we had not seen earlier. The earlier styles where of assignment where we will say top_ =1, or we will say data = new char[10] like this. But here this special way of writing the initialization which is possible only when you are using a constructor and note that after the signature of the constructor you have a separator as colon and you need to write this whole list of initializations of data members between this colon and the opening curly brackets of the constructor body.

Here you write the data member one after the other separated by coma and after every data member within a pair of parenthesis you write what value you want to initialize them with. Certainly, every data member can occur here only once and it is not necessary

that all data members will have to be initialized, but we can initialize all of them also if we want as we done in this case.

What will happen for this particular version of the stack? As soon as the control passes this point, where the stack as being instantiated this constructor will get called, so for the stack object s the top will get initialize to -1 there will be a dynamic allocation of the character array of size 10 and its pointer will be set to data, and then the stack call, this message will get printed and on completion the control will come back to this point where the string reverse will proceed.

With this we can see that the constructor can be used to initialize the data member in multiple different ways and the mechanism of C++ will ensure that instantiation itself is initialization of the object as well. And the application has no over head of remembering or trying to make sure that the initialization happens for the stack data part or for the stack top marker part, it will be the responsibility of the constructor and the compiler will call this constructor implicitly we do not have to remember to call it, at an appropriate point it will get called every time.

(Refer Slide Time: 16:44)

Constructor	Member Function
<ul style="list-style-type: none"> ● Is a member function with <code>this</code> pointer ● Name is same as the name of the class ● <code>class Stack { public:</code> ● <code>Stack();</code> ● <code>};</code> ● Has no return type ● <code>Stack::Stack(); // Not even void</code> ● No return; hence has no return statement ● <code>Stack::Stack(); top,-1;</code> ● <code>{} // Returns implicitly</code> <ul style="list-style-type: none"> ● Initializer list to initialize the data members ● <code>Stack::Stack() : data_(new char[10]), // Init data_</code> ● <code>top_(-1) // Init top_</code> ● <code>{}</code> ● Implicit call by instantiation / operator <code>new</code> ● <code>Stack s; // Calls Stack::Stack()</code> ● May have any number of parameters ● Can be overloaded 	<ul style="list-style-type: none"> ● Has implicit <code>this</code> pointer ● Any name different from name of class ● <code>class Stack { public:</code> ● <code>int empty();</code> ● <code>};</code> ● Must have a return type ● <code>int Stack::empty();</code> ● Must have at least one return statement ● <code>int Stack::empty()</code> ● <code>{ return (top_ == -1); }</code> ● <code>void pop()</code> ● <code>{ --top_; } // Implicit return</code> ● Not applicable <div style="text-align: center;"> </div> <ul style="list-style-type: none"> ● Explicit call by the object ● <code>s.empty(); // Calls Stack::empty()</code> ● May have any number of parameters ● Can be overloaded

NPTEL MOOCs Programming in C++

Partha Pratim Das

7

Now, just to sum up what does a constructor do, a constructor as we have seen is necessarily a member function, but it has a number of differences with the member function. So, being a member function, it has a this pointer like any member function, but its name as I already mentioned is specifically the name of the class which is not so for the other member functions.

You may have noticed already that there is no return type being shown because the job of the constructor is to construct the object and initialize it, it is not meant for doing a computation and giving you a value back. Even though at the constructor is a function it does not return any value and it does not have a return type not even void. You have seen function which have void return type which say that I am not returning anything, but in constructor that is not the case. In the constructor what you say is there is no mechanism to return any value at all, so there is no return type to be specified.

Since there is no return type there is no return statement either so you will not find any return statement unlike a common member function where there can be return statement or the return could be implicit if your return type is void, but a constructor will not need any return statement it will return implicitly. In addition, the constructor will have the initialization list which I have just explained the list of data members within parenthesis the initialization value. The list starts with a colon ends with the beginning of the constructor body, and this is something which is not parallel for any other member functions. The constructor does not have any explicit call it is implicitly called by instantiation as soon as I instantiate this object the constructor will get called, whereas the call objects are necessarily explicit in nature.

We will see examples, but finally a constructor may have any number of parameters in the example that we have seen right now there is no parameter, but a constructor like any other member function may have any number of parameters, and like other member functions a constructor can be overloaded as well.

(Refer Slide Time: 19:20)

The slide title is "Program 13.06: Complex Parameterized Constructor". On the left, there is a sidebar with navigation links: Module 13, Partha Pratim Das, Objectives & Outline, Constructor, Parameterized Constructor, Default Constructor, Object Lifetime, Automatic Storage Duration, Summary, and Log in. Below the sidebar is a small video thumbnail of a man speaking. The main content area contains the following C++ code:

```
#include <iostream>
using namespace std;

class Complex { private: double re_, im_;
public:
    Complex(double re, double im) : re_(re), im_(im) {} // Ctor w/ params
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() {
        cout << "(" << re_ << "j" << im_ << ")" << endl;
    }
};

int main() {
    Complex c(4.2, 5.3); // Complex::Complex(4.2, 5.3)
    Complex d = { 1.6, 2.9 }; // Complex::Complex(1.6, 2.9)

    c.print();
    d.print();

    return 0;
}

=====
[4.2+5.3j] = 6.7624
[1.6+2.9j] = 3.3121
```

At the bottom right of the slide, there is a small circular icon with a play symbol.

A constructor is a special function which basically will help us in initializing objects implicitly all the time. We quickly flip through a couple of slides looking at variety of constructors that you can define in C++, you can have parameters in the constructor these are called Parameterized constructor. We are showing the double type again which we have already seen so this parameter values can be used to initialize the data members through the initialization list. Then the constructor will get invoked implicitly by instantiation and there are two syntax typically that can be used for it.

One is a typical function call kind of syntax, where you put the object name and with in parenthesis you put the parameters of the constructor in the same order in which they are define for the constructor. So when I do write this particular object instantiation it means that when control passes this point the constructor of complex will get called where 4.2 will get go as the value of re, 5.3 will go as a value of im and with that the object will get constructed and set as the object c. Beyond this if you do c.re you should get a value 4.2, that is what this print statement show here, where we print the complex number complex notation and compute its norm. Similarly, if I do im component of c I will get 5.3.

In the second, in case of d we just show the same thing in a different alternate notation this is called list initialization. That is, if you have multiple parameters in a constructor

you could put the initial values of these parameters in terms of a pair of curly braces (Refer Time: 21:22) in the list notation and use an initialization symbol after the object name. The effect of both are same, they are just alternate notations for doing the same operation. So, constructors which have one or more parameters is known as Parameterized constructors.

(Refer Slide Time: 21:45)

Now, constructor in every respect is just another member function, or specifically just another function in C++ therefore, if I have parameters I can also have default values for parameters. So, I can have constructor with different default values so you show an example here again with the two parameters `re` and `im` having default values 0.0. Then based on all the rules of default parameter for functions and default parameters for invocation of functions, we can use this same constructor to construct objects in three different ways I can put both the parameters I can just specify the first parameter or I can specify none of the parameter. Parameters that are not specified will take the default values and accordingly if you print out you will be able to see the corresponding objects as they have got created. It is just showing that the whole mechanism of defaulting parameter values all rules has we have learnt for function with default parameters will also apply to constructors.

(Refer Slide Time: 23:06)

Module 13
Partha Pratim Das
Objectives & Outline
Constructor Parameterized Overloaded
Destructor
Default Constructor
Object Lifetime
Automatic Static Dynamic Summary

Program 13.08: Stack: Constructor with default parameters

```
#include <iostream>
using namespace std;

class Stack { private: char *data_; int top_;}
public:
    Stack(size_t s = 10); // Size of data_ defaulted
    int empty() { return (top_ == -1); }
    void push(char x) { data_[++top_] = x; }
    void pop() { --top_; }
    char top() { return data_[top_]; }
};

Stack::Stack(size_t s) : data_(new char[s]), // Array of size s allocated
                        top_(-1)
{ cout << "Stack created with max size = " << s << endl; }

int main() {
    char str[] = "ABCDE";
    Stack s(strlen(str)); // Create a stack large enough for the problem

    for (int i = 0; i < s.size(); ++i) s.push(str[i]);
    while (!s.empty()) {
        cout << s.top(); s.pop();
    }
    return 0;
}
/*
Stack created with max size = 5
ABCDE

```

NPTEL MOOCs Programming in C++ Partha Pratim Das 10

This is just another example here, as we are going back to stack so we show a constructor for a stack with a default size_t value 10 (size_t = 10). In the definition of the constructor we have a parameter, so if I want I can pass this parameter value for example here, we are constructing the object taking the length of the string that we want to reverse because we know that if this stack is meant to reverse the given string STR then it is never going to need more size than the length of the string so we can use that and make a stack which is just large enough to accommodate the string and here you can see that the constructor is printing as to what is the size of the stack that has got created. If I do not use this, if we do not provide all this information of what should be the size if we just write Stack s then certainly it will take the default parameter value 10 and create a stack of 10 elements.

So, these are the different examples of constructors that we can have.

(Refer Slide Time: 24:25)

The slide title is "Program 13.09: Complex Overloaded Constructors". On the left, there is a vertical sidebar with navigation links: Module 13, Partha Pratim Das, Objectives & Outline, Constructor Parameterized Overloaded Constructor, Default Constructor, Object Lifetime Automatic Static Dynamic Summary. The main content area contains the following C++ code:

```
#include <iostream>
using namespace std;

class Complex { private: double re_, im_; public:
    Complex(double re, double im): re_(re), im_(im) {} // Two parameters
    Complex(double re): re_(re), im_(0.0) {} // One parameter
    Complex(): re_(0.0), im_(0.0) {} // No parameter

    double norm() { return sqrt(re_*re_ + im_*im_); }

    void print() { cout << "(" << re_ << "j" << im_ << ")" << " = " << norm() << endl; }

};

int main() {
    Complex c1(4.2, 5.3), // Complex::Complex(4.2, 5.3)
            c2(4.2), // Complex::Complex(4.2)
            c3(); // Complex::Complex()

    c1.print();
    c2.print();
    c3.print();

    return 0;
}

=====
[4.2+5.3j] = 6.7624
[4.2j0] = 4.2
[0j0] = 0
```

At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

Next like any other C++ group of functions the constructors can also be overloaded, which mean that a class can have more than one constructor because necessarily being a constructor it has the same name as the class name, so if I want to write two constructor they will necessarily have the same name, but it is permitted to do so as long as they differ in the number of parameters or types of parameters and so on. All rules of function overloading as we are learnt will apply exactly in the same way in the terms of over-loaded constructor. So here we are showing the three constructors for complex which one which takes two parameters re and im both, one which just takes array and one which takes nothing.

And we are using these parameters in terms of the three possible instantiations of the number of parameters deciding which particular form of the constructor that will get invoked. For example, if we look at this instantiation of c2, when c2 will gets instantiated then naturally this constructor which has one parameter will get invoked. Whereas, if c1, when c1 is being instantiated then the constructor having two parameters will get invoked, so all rules of over loaded constructors apply in this case.

In this way constructors can be over loaded and we can write construction process in terms of variety of different parameters and their combinations and so on and as we go

forward we will see several more examples of how overloaded constructions becomes important for writing very effective construction mechanism for the different classes that we built.

Programming in C++
Prof. Partha Prathim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 24
Constructors, Destructors and Object Lifetime (Contd.)

Welcome to module 13, part-2 of Programming in C++.

In the first part of this module, we have seen how objects can be constructed using the constructor of classes, how implicitly constructors get called and they can be used to initialize data members of objects by specifying through the initialization list, and we have also seen how we can have parameterized constructors? How we can use default values for those parameters? And how we can have overloaded constructors for all varieties of construction and initialization mechanism to be provided?

(Refer Slide Time: 01:16)

 Module 13 Partha Pratim Das Objectives & Outline Constructor Parameterized Overloaded Destructor Default Constructor Object Lifetime Automatic Static Dynamic Summary	<h3 style="text-align: center;">Program 13.10/11: Stack: Destructor</h3> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50%; vertical-align: top; padding: 5px;"> Automatic Array <pre>#include <iostream> using namespace std; class Stack { private: char *data_; int top_; // Dynamic public: Stack(); // Constructor void de_init() { delete [] data_; } // Stack methods }; Stack::Stack(): data_(new char[10]), top_(-1) { cout << "Stack::Stack() called\n"; } int main() { char str[10] = "ABCDE"; Stack s; // Init by Stack::Stack() call // Reverse string using Stack de_init(); return 0; } ----- Stack::Stack() called EDCBA</pre> </td> <td style="width: 50%; vertical-align: top; padding: 5px;"> Dynamic Array <pre>#include <iostream> using namespace std; class Stack { private: char *data_; int top_; // Dynamic public: Stack(); // Constructor ~Stack(); // Destructor // Stack methods }; Stack::Stack(): data_(new char[10]), top_(-1) { cout << "Stack::Stack() called\n"; } Stack::~Stack() { cout << "\nStack::~Stack() called\n"; delete data_; } int main() { char str[10] = "ABCDE"; Stack s; // Init by Stack::Stack() call // Reverse string using Stack return 0; } // De-init by Stack::~Stack() call ----- Stack::Stack() called EDCBA Stack::~Stack() called</pre> </td> </tr> <tr> <td colspan="2" style="text-align: right; padding-top: 5px;"> • Dynamically allocated data_ leaks unless released before program loses scope of s • Application may forget to call de_init(); Also, when should de_init() be called? <small>NPTEL MOOCs Programming in C++</small> </td> </tr> <tr> <td colspan="2" style="text-align: right; padding-top: 5px;"> • Can de-initialization (release of data_) be a part of scope rules? • Yes. Destructor is implicitly called at end of scope </td> </tr> </table> <p style="text-align: right; margin-top: -10px;">Partha Pratim Das 12</p>	Automatic Array <pre>#include <iostream> using namespace std; class Stack { private: char *data_; int top_; // Dynamic public: Stack(); // Constructor void de_init() { delete [] data_; } // Stack methods }; Stack::Stack(): data_(new char[10]), top_(-1) { cout << "Stack::Stack() called\n"; } int main() { char str[10] = "ABCDE"; Stack s; // Init by Stack::Stack() call // Reverse string using Stack de_init(); return 0; } ----- Stack::Stack() called EDCBA</pre>	Dynamic Array <pre>#include <iostream> using namespace std; class Stack { private: char *data_; int top_; // Dynamic public: Stack(); // Constructor ~Stack(); // Destructor // Stack methods }; Stack::Stack(): data_(new char[10]), top_(-1) { cout << "Stack::Stack() called\n"; } Stack::~Stack() { cout << "\nStack::~Stack() called\n"; delete data_; } int main() { char str[10] = "ABCDE"; Stack s; // Init by Stack::Stack() call // Reverse string using Stack return 0; } // De-init by Stack::~Stack() call ----- Stack::Stack() called EDCBA Stack::~Stack() called</pre>	• Dynamically allocated data_ leaks unless released before program loses scope of s • Application may forget to call de_init(); Also, when should de_init() be called? <small>NPTEL MOOCs Programming in C++</small>		• Can de-initialization (release of data_) be a part of scope rules? • Yes. Destructor is implicitly called at end of scope	
Automatic Array <pre>#include <iostream> using namespace std; class Stack { private: char *data_; int top_; // Dynamic public: Stack(); // Constructor void de_init() { delete [] data_; } // Stack methods }; Stack::Stack(): data_(new char[10]), top_(-1) { cout << "Stack::Stack() called\n"; } int main() { char str[10] = "ABCDE"; Stack s; // Init by Stack::Stack() call // Reverse string using Stack de_init(); return 0; } ----- Stack::Stack() called EDCBA</pre>	Dynamic Array <pre>#include <iostream> using namespace std; class Stack { private: char *data_; int top_; // Dynamic public: Stack(); // Constructor ~Stack(); // Destructor // Stack methods }; Stack::Stack(): data_(new char[10]), top_(-1) { cout << "Stack::Stack() called\n"; } Stack::~Stack() { cout << "\nStack::~Stack() called\n"; delete data_; } int main() { char str[10] = "ABCDE"; Stack s; // Init by Stack::Stack() call // Reverse string using Stack return 0; } // De-init by Stack::~Stack() call ----- Stack::Stack() called EDCBA Stack::~Stack() called</pre>						
• Dynamically allocated data_ leaks unless released before program loses scope of s • Application may forget to call de_init(); Also, when should de_init() be called? <small>NPTEL MOOCs Programming in C++</small>							
• Can de-initialization (release of data_) be a part of scope rules? • Yes. Destructor is implicitly called at end of scope							

Now, we will look at the other side of the story. We will look at the process of what happens when an object reaches the end of its lifetime or it is time to destruct. So, again we go back to the stack example. We are looking into the stack having private data and the container that contains the elements of the stack is a dynamically allocated array therefore, we just have a pointer.

So, in terms of the stack we know what needs to be done, in terms of the construction we

know what needs to be done, we have seen this. A dynamically allocation is done with operator new for an array of 10 characters and that pointer is stored in data.

So, the stack will work, it will initialize through this construction and then it will do the reverse. So, just for clarity of presentation I have skipped all the details of the stack methods and the reversing string code they are same as before, but having done that when we reach to this point, we are done with this Stack and we should return. But the point is, as we have already noted there is a dynamically allocated array whose pointer is held by the data component of the stack object s.

Now, if we return at this point; we return from main and the control goes away, then certainly this dynamically created array remain becomes inaccessible because quiet clearly s, the object s that is instantiated here is available only within the scope of this function. So, if I do return and go out of it then there is no way that I can access s, there is no way that I can therefore access s.data or release the memory that we have actually acquired through the new process done in the construction. So, to avoid this problem to be able to manage our resources right and release them, whenever we are done with using the resource there has to be a matching mechanism of De-Initialization, which will undo what we had done at the initialization time.

So, we make a similar assumption as we are done earlier for initialization, let there be a function de_init(), why do we need a function? And why do we need that function to be put here? Because to release the allocated array, we need to access the “data_” data member of the stack class which is private. So, we put de_init() method in the public and then call that method, style similar to what we did in initialization and the problems this will cause the problems which is similar to what happened for initialization also. That is precisely what if the application would forget to call the init that is one part.

Second, in this case we have a bigger problem because it is not only about forgetting to call de_init(), but there is a bigger issue is even if I remember, what is the exact location where I should call de_init(), if I call de_init() and after calling de_init(), I try to use the stack. Then I will have evaluation because the stack does not have the container any more. I have already released it, but de_init() is within the same scope where the stack is defined. So, it is possible that after calling de_init() here may try to use the stack. So, I have to be very careful that every call to de_init() must ensure that after that there is no

possible use of s.

Second from a function; this is a very simple case we are showing here, from a function there could be multiple places from where I actually return, if I am returning from multiple places I do not know before and which particular return statement will be taken by the control flow. So, I will need to remember and put `de_init()` at each one of these places. So, it is a complete mess, it is a complete mess and its is one of the major reasons is a known source of problem in C that resources allocated are not properly de-allocated or not properly released.

In C++, we have a beautiful solution in terms of what is known as destructor. Let us look at that solution; what we do, we introduce another member function in the class. This member function too have the same name as the class, but the difference being it is prefixed to with the special symbol tilde (~) and it is called the destructor of the class. So, what happens is as at the time of object instantiation, the constructor is getting called.

Similarly, when the object goes out of scope here that is, this is the curly brace on which the object becomes goes out of scope in the sense that after this curly brace has been passed by the control, there is no way to talk about this s that is a scope rule of C, that is the scope rule of C++. So, right at this point when it is trying to cross the end of this scope, a call will be made to the destructor of the class for this object s.

At this point `s.~Stack()`, this function will get called and the beauty of the whole thing is that the user does not, or the programmer does not need to remember and make the call. The compiler implicitly would compute that, this is the end of scope for s and implicitly put this call at the end of the scope. So, if we now look at the whole mechanism together we do not need this kind of `de_init()` is no more, is not required here.

The stack gets initialized at this point to the constructor call to the call to this function and that is the message that it prints. Then the stack is used to reverse the string. The reverse string is output and then the control goes out of scope at this point and implicitly the destructor function is called and you can make out that the destructor function has been called by finding out that the message within the destructor function has been printed at the output.

So, this ensures that the data that was dynamically held up in this data member of the

stack class can be released at this point. So, whenever no matter in what context I use the stack, whenever I instantiate a stack, the constructor ensures that proper allocation will be done to the data, proper initialization will happen to stop and whenever that instantiated stack object gets out of scope, the compiler will also ensure that the constructor is called so that the proper release of the allocated data can happen. So, this combined pair of constructor and destructor gives us a completely well defined structured way of managing the life time of every object in a very clean manner in C++.

(Refer Slide Time: 09:53)

Destructor	Member Function
<ul style="list-style-type: none"> ● Is a member function with <code>this</code> pointer ● Name is ~ followed by the name of the class <code>class Stack { public:</code> <code> ~Stack();</code> <code>}</code> ● Has no return type <code>Stack::~Stack(); // Not even void</code> ● No return; hence has no return statement <code>Stack::~Stack()</code> <code>{} // Returns implicitly</code> ● Implicitly called at end of scope or by operator <code>delete</code>. May be called explicitly by the object (rare) <code>{</code> <code> Stack s;</code> <code> // ...</code> <code>} // Calls Stack::~Stack(&s)</code> ● No parameter is allowed - unique for the class ● Cannot be overloaded 	<ul style="list-style-type: none"> ● Has implicit <code>this</code> pointer ● Any name different from name of class <code>class Stack { public:</code> <code> int empty();</code> <code>}</code> ● Must have a return type <code>int Stack::empty();</code> ● Must have at least one return statement <code>int Stack::empty()</code> <code>{ return (top_ == -1); }</code> ● Explicit call by the object <code>s.empty(); // Calls Stack::empty(&s)</code> ● May have any number of parameters ● Can be overloaded

NPTEL MOOCs Programming in C++

Partha Pratim Das 13

So, to look at formal properties of destructor as we have already seen destructor is also a member function. It has at this pointer like any other member function, its name is special, it is a tilde followed by the name of the class, like the constructor destructor too does not have a return type because certainly we have seen it is called at the end of the scope and therefore, if it were to return something then there is no taker for that returned value. There is no meaning of having a return computed from the destruction of an object, so destructors also do not have any return type - not even void.

Certainly, consequently there is no return statement in a destructor and as we have already seen which is the most significant and important part is the destructors are called implicitly at the end of the scope and can be used seamlessly for any automatic object.

There are other ways to call the destructor also, we will illustrate that, but usually always the call to a destructor is implicit by the compiler through the measurement of this

automatic scope in which the particular object was constructed. In contrast to constructor, a destructor is not allowed any parameter. Since, it is not allowed, any parameter you will understand by now that the consequence of that is exactly, you cannot overload the destructor. So, if you cannot have any parameter and if you cannot overload the destructor which means that the destructor of a class is unique.

So, the scenario is a class may have one or more constructors as many as it wants through different mechanisms of overloading, but it must have only one way to destruct and there is a profound reason that the things are defined this way because obviously, once when you are about to construct an object, you certainly have choice, you would like choice in terms of what kind of object what initial values for parameters and all that, you want to set and based on that it is quiet logical that you may need different mechanisms or different parameters sets to construct the object and therefore, you need a number of overloaded constructors.

But, once we have constructed the object, there is no trace of the fact is to which constructor was used to construct it is object. There is no trace of whether a data member was initialized or it was subsequently set a value and so on. Therefore, when you want to destruct, all objects are similar. All that he as saying that I am done with this object, I want to release all the allocated resources that are held by my data members and I want to free up the memory in which the object currently resides. So, I need to comply with the destruction process and therefore, the destruction process for an object in a class has to be unique and therefore, there is unique single destructor for every class in the C++ design.

(Refer Slide Time: 13:17)

The slide has a blue header bar with the title "Default Constructor / Destructor". On the left, there is a sidebar with a logo at the top, followed by "Module 13", "Partha Pratim Das", "Objectives & Outline", and a list of topics: "Constructor", "Parameterized", "Overloaded", "Destructor", "Default Constructor", "Object Lifetime", "Automatic", "Static", "Dynamic", and "Summary". The main content area contains two bullet points: "Constructor" and "Destructor", each with a list of sub-points. At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" with the number 14.

- Constructor
 - A constructor with no parameter is called a *Default Constructor*
 - If no constructor is provided by the user, the compiler supplies a *free default constructor*
 - Compiler-provided (default) constructor, understandably, cannot initialize the object to proper values. It has no code in its body
 - Default constructors (free or user-provided) are required to define arrays of objects
- Destructor
 - If no destructor is provided by the user, the compiler supplies a *free default destructor*
 - Compiler-provided (default) destructor has no code in its body

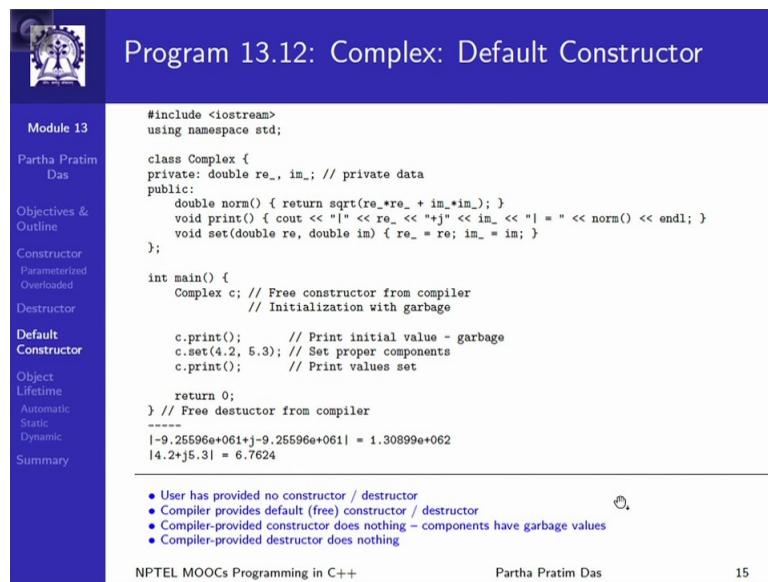
At this point, once we have seen the constructor and destructor, let us also note that constructors can be default. A constructor which has no parameter is called a default constructor. So, user actually has two options, user can write a constructor or it may not write a constructor, but the C++ has a mechanism must have a constructor for every class. So, what the compiler does is, if the user provides a constructor it will use that constructor, but if the user does not provide a constructor for a class then the compiler will supply a free default constructor.

Now, if the compiler supplies a constructor, naturally the compiler do not know what parameters you might want. The compiler does not know how to initialize your data members. So, just to make the whole process coherent, the compiler will give you a free default constructor and the code will compile assuming that you have a free default constructor, you have default constructor. So, it is usually good to provide a constructor even if it does not have a parameter because once you provide a constructor, even if it is default, your constructor will be used and compiler will not provide the free constructor because if you write the default constructor or if you write any other overloaded constructor then you really know what are you are initializing.

If the compiler provides it, then it will just have a whole lot of garbage values for your data members possibly, but the mechanisms exist that the compiler will give you a free default constructor. Similarly for destruction, if the user has not provided a destructor

then the compiler will supply a free default destructor which certainly does nothing it has got a empty body, but there is no code in their body because that compiler does not know what possibly needs to be released or if it is nothing to be released, but to make the whole mechanism work the compiler will indeed provide a free default destructor.

(Refer Slide Time: 15:42)



Program 13.12: Complex: Default Constructor

Module 13
Partha Pratim Das
Objectives & Outline
Constructor Parameterized Overloaded Destructor
Default Constructor
Object Lifetime Automatic Static Dynamic Summary

```
#include <iostream>
using namespace std;

class Complex {
private: double re_, im_; // private data
public:
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "(" << re_ << "+" << im_ << ")" = " << norm() << endl; }
    void set(double re, double im) { re_ = re; im_ = im; }
};

int main() {
    Complex c; // Free constructor from compiler
    // Initialization with garbage
    c.print(); // Print initial value - garbage
    c.set(4.2, 5.3); // Set proper components
    c.print(); // Print values set

    return 0;
} // Free destructor from compiler
-----
|-9.25596e+061+j-9.25596e+061| = 1.30899e+062
|4.2+j5.3| = 6.7624
```

- User has provided no constructor / destructor
- Compiler provides default (free) constructor / destructor
- Compiler-provided constructor does nothing – components have garbage values
- Compiler-provided destructor does nothing

NPTEL MOOCs Programming in C++ Partha Pratim Das 15

So, here we show an example of default constructor. Please take a look into this class. It has our complex class, the old friend of us. What I have added for illustration, I have added a new member function method set which takes two double values and sets two values into the two data members of the class which can be invoked at any point of time to set a complex value to my class, but what is missing now, you know is that there is no constructor and there is no destructor provided. So, what will happen if we; when this instantiation is happening at this point there will be a call to the default constructor that has freely been provided by the compiler?

So, this call will happen even though no such function has been defined by you in the body of the class. So, what will this constructor initialize re and im with, it does not know what it needs to be initialized and therefore, it will just leave some garbage values at these points. So, to understand that, if after this object has been constructed if you immediately after that if you print this object. I mean this is just one case, if you try this experiment yourself, you will get some different values possibly. So, just prints this which is some garbage bit pattern which existed where re and im should be there, but

once I have used the set function that I have provided here to set the component values to re and im and then I do the print, I do get the proper values again.

Simple advice that, if you do not provide the constructor then unfortunately the compiler will not give an error, compiler will provide a default constructor default destructor and go ahead with that. So, he will run the risk of not having proper values, proper initialization syntax. So, whenever you write a class, make sure that you write a constructor and the destructor.

(Refer Slide Time: 18:03)

```
#include <iostream>
using namespace std;

class Complex { private: double re_, im_;
public:
    Complex(): re_(0.0), im_(0.0) // Default Ctor
    {
        cout << "Ctor: (" << re_ << ", " << im_ << ")" << endl;
    }
    ~Complex() // Dtor
    {
        cout << "Dtor: (" << re_ << ", " << im_ << ")" << endl;
    }
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "|" << re_ << "j" << im_ << "|" = " << norm() << endl; }
    void set(double re, double im) { re_ = re; im_ = im; }
};

int main()
{
    Complex c; // Default constructor -- user provided

    c.print(); // Print initial values
    c.set(4.2, 5.3); // Set components
    c.print(); // Print values set

    return 0;
} // Destructor
-----
Ctor: (0, 0)
|0+0j| = 0
|4.2+5.3j| = 6.7624
Dtor: (4.2, 5.3)

• User has provided a default constructor
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 16

Here is another example where we are showing the default constructor, but in this case the only difference is the default constructor is not provided by the compiler, but the default constructor is provided by the user. So, user has written this. So, when you do the first printing after the initialization naturally, since there was an initialization with proper values 0 and 0, you get that, there is proper initial values and not the garbage kind of garbage that we saw in the last example.

Similarly, a destructor is also given here. In this case, the destructor prints this message. In reality a destructor for a complex class may not do anything, but it is always good to provide one with a blank body.

With this we will close on the construction and destruction process. We have learnt that every class will have a destructor which is unique and which will get invoked for

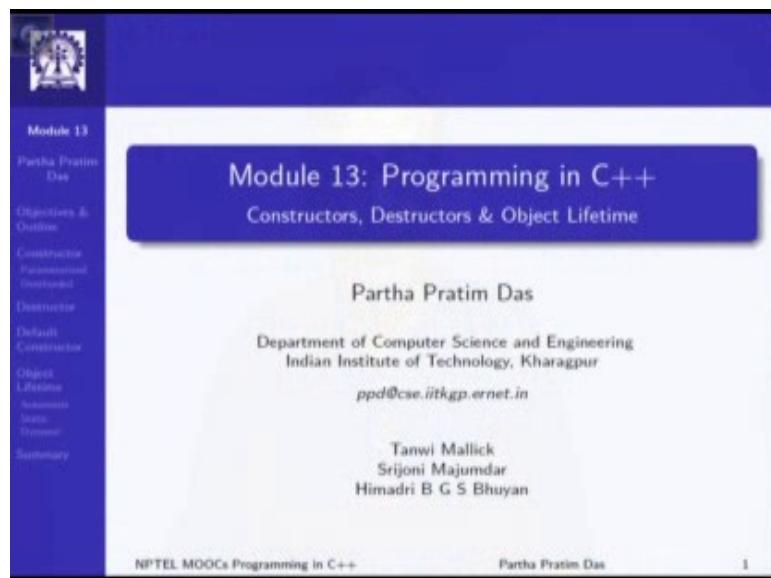
automatic objects, it will get invoked at the end of this scope and within this destructor, we can release clean up any kind of resources that we have been holding on to, and we have also seen that the compiler provides a default constructor and a default destructor provided the user has not written a destructor or a constructor for the class.

Programming in C++
Prof. Partha Prathim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 25
Constructors, Destructors and Object Lifetime (Contd.)

Welcome to part 3 of module 13.

(Refer Slide Time: 00:24)



Module 13: Programming in C++
Constructors, Destructors & Object Lifetime

Partha Pratim Das
pppd@cse.iitkgp.ernet.in

Tanwi Mallick
Srijoni Majumdar
Himadri B G S Bhuyan

NPTEL MOOCs Programming in C++
Partha Pratim Das

In this module, we have already discussed about object construction and destruction through variety of constructors and the destructor. We have talked about the free constructor and destructor that the compiler may provide. Equipped with all these, now we will take a deeper look into what is known as object lifetime. Where, we will answer the basic question of when an object is ready and how long it can be used.

(Refer Slide Time: 00:58)

**Object Lifetime: When is an Object ready?
How long can it be used?**

Application	Class Code
<pre>void MyFunc() // E1: Allocation of c on Stack { ... Complex c; // E2: Ctor called ... c.norm(); // E3: Use ... return; // E7: Dtor called } // E9: De-allocation of c from Stack</pre>	<pre>Complex::Complex(double re = 0.0, // Ctor double im = 0.0); re_(re), im_(im) // E3: Initialization { // E4: Object Lifetime STARTS cout << "Ctor:" << endl; } double Complex::norm() // E6 { return sqrt(re_*re_ + im_*im_); } Complex::~Complex() // Dtor { cout << "Dtor:" << endl; } // E9: Object Lifetime ENDS</pre>
Event Sequence and Object Lifetime	
<pre>E1 MyFunc called. Stackframe allocated. c is a part of Stackframe E2 Control to pass Complex c. Ctor Complex::Complex(&c) called with the address of c on the frame E3 Control on Initializer list of Complex::Complex(). Data members initialized (constructed) E4 Object Lifetime STARTS for c. Control reaches the start of the body of Ctor. Ctor executes E5 Control at c.norm(). Complex::norm(&c) called. Object is being used E6 Complex::norm() executes E7 Control to pass return. Dtor Complex::~Complex(&c) called E8 Dtor executes. Control reaches the end of the body of Dtor. Object Lifetime ENDS for c E9 return executes. Stackframe including c de-allocated. Control returns to caller</pre>	

NPTEL MOOCs Programming in C++ Partha Pratim Das 17

So, I will start with a basic abstract chart of an application. So, on the left-hand side is my application, which is, let say is one function MyFunc, which does not do anything meaningful. It just a function MyFunc and on the right at snippets of the class code for my complex class, they have necessarily three codes. That I have picked up here; the constructor code, a member function called norm() which finds a norm of the number and the destructor code.

So, if we look into what is going to happen? From the instant, the function MyFunc is called by some caller, till the time MyFunc completes its execution and returns the control to the caller. Let us try to see what happens to the objects that are defined in this scope of MyFunc. These are objects like Complex c, which are defined locally within this function scope. These are commonly called automatic objects. Now, to trace the sequence of events that happens. In the comments, you will see that I have annotated the comments with some event number E1, E2, like this. These numbers show the order in which the events happen.

So, the first event is E1, where the function has been called; which means as soon as the function has got called, there is a stack frame allocated on the stack. Corresponding to this function and this local variable, this local object c has an allocation on that stack frame, which is, which will give its memory location which eventually will become this pointer. So, this is what happens at E1 and then the control moves on. There is something

in between, which we do not care. And, it comes to the point where the control is about to pass the instantiation of c, when the second event E2 happens; that is, the constructor gets called for this class; which means that the control goes over to here with the address of the c object as allocated on the stack frame.

As it reaches the constructor, first the initialization has to happen; which means that before the body of the constructor can even start executing, all data members as listed in the initialization list will one after the other be initialized with the value specified. So, first the re gets initialized, then im gets initialized and this is the event three, E3, that will happen. On completion of this initialization, the control will reach the beginning of the body of the construction. And at this point; precisely at this point, we say that the object is actually available. Its lifetime starts. And that is the event E4.

So, if you look into the table below where, whatever I am saying is written down here in the table below. This is the beginning of the life time. So, please bear in mind that the object does not start or its lifetime does not start, when the constructor is called away, the instantiation happens. But, it starts when the constructor has been called and it has completed the initialization part of it.

So, when the lifetime of an object starts, the basic notion is the object has all its data members completely ready, and with initial values as specified in the constructor. Before that, from the time E2 happens and to the time E4 happens, that is, during E3 the object has what is known as an inconsistent state. It may or may not have proper values of the data members. For example, it is quite possible that if I look into the object during this time, maybe re component has been initialized to the value given and im component is yet to be initialized. So, the object is said to be in an inconsistent state, as long as the initialization is going on. And, only on completion of that the control reaches the beginning of the constructor body. The event marked as E4, where the object is ready and can now be used.

So, in the body of the constructor, actually you write any code, which can assume that the object is already properly ready and going. So, once the construction is over, this constructed body, the control comes back here. Some more stuff keeps on happening and then, again this object is used to invoke some method of the complex class, which is event E5; which gives a call to this method. This execution start; which is event E6 then

it computes the norm of the number and returns that norm back here and again things continue.

So, this basically, if you look into this part in the middle after the construction has happened, then we may have several different use of the object, where the object, where data members are being used, the data members are being changed, different methods are getting invoked and so on and so forth. The object is ready and is getting used. It is in the prime of its lifetime, till it reaches the return statement.

When it reaches the return statement, then you know that at this return statement, you know that as soon as this return statement will get executed, the control will go out of MyFunc. And, if the control goes out of MyFunc, this scope in which this object c belongs will no more be available. When the control goes out of MyFunc, the allocation of the frame on the stack for this function will no more be valid. Therefore, the address of c on the stack will no more be valid. So, this is the point.

Precisely, the precise point is quiet delicate. It is right before the return, but at the return it is not the previous statement, is not the next statement. But, right at this point the destructor will have to get called. So, here a call to c. \sim Complex() will get called. So, on E7, the control will go to the destructor. The destructor will go through. The destructor body could do different d initialization activities. But, please remember the object is still in use. The object still has got a life time which is valid, till it reaches the end of the destructor body.

Event E8, when it is considered that the lifetime of the object is over and the control comes back to return again. So, when the control comes back, at that point the object is no more a valid one. The object lifetime is already over and then proceeds with the purpose of return, which will return the control back to the caller. And, it will also deallocate the stack frame for this function. A part of which was containing the object c, which we were tracking the lifetime for.

So, if we just summarize, it is this point in the construction and it is this point in the destruction. During the execution of the program, which defines the lifetime of an object in the particular context in which we are showing we will show different lifetime constructs; we will show how lifetimes differ. But, it is always between the beginnings of the body of constructor to the end of the body of destructor; is a time during which the

corresponding object is alive and is considered to be its lifetime.

Actually, this understanding will also help us to understand something more specifically about the initialization list. I often get questions from people trying to learn the construction, understand the construction process that why do I need an initialization list? I could have instead in the body of the constructor written something like `re_ = re` or `im_ = im`. I could have written this as a part of the body of the constructor, instead of doing the initialization here.

So, there are two questions; as to why the initialization list is at all required? And, even if the initialization list is provided, is it necessary to initialize? The answer is as you have understood the object is the moment the control reaches this point; the object is considered live; the object is constructed. The construction is complete.

So, if you put the values of the data members within the body of the constructor, then at this point when the lifetime starts, your data members have inconsistent values. It may have typically garbage values. Now, if you look into the example of a complex class which has just two doubles, it really is not going to matter in your course of program execution as to whether you really, truly initialize or you do not initialize and let the garbage value of `re` and `im`, start the object, and then is a body of the constructor. You set new values to `re` and `im`. It is not going to make a difference, but later on we will show examples where it does really matter as to whether you have a proper object when you reach this point; this is the beginning point of the object. When you reach this point, whether you have a properly initialized object or not for many classes that may make a difference. And when it does, then you really have a problem because there is no way to solve that other than using the initialization list. This is another factor to deal the initialization list. Just think of that there are number of data members in an object. And, in the initialization list or in the process of initialization, I can initialize; I mean whatever order I want. So, if initialization of one data member is dependent on the initialization of the other, it will matter as to which data member is initialized earlier and which is initialized later.

Now, if I write the initialization as assignments in the body of the constructor, then there could be; if there are n different data members, there could be factorial n different ways of initializing data members. But, if I write it as an initialization list, the compiler

follows a unique approach. The compiler initializes them in the order in which you write them in the class, not in the order in which you write them in the initialization list. So, for this example of complex, we have written it as re and im. We could have written this as; we could have written im initialization first, followed by re initialization. But, even if I write the initializer list like this, this initialization will happen before this initialization, as long as the data member re precedes the data member im in the definition of the class.

So, the result of having initialization list is without the user having to put any effort, the process of initialization is unqualified, in the sense that data members are necessarily initialized from top to bottom, a mechanism which you could not have been able to guarantee, if you were to put initial values in the body of the constructor. So, with this we understand that this is what the lifetime is and this is where it starts and this is where it ends. In the next couple of slides, we will quickly take look into different examples of lifetime and try to understand this better.

(Refer Slide Time: 15:31)

The slide has a blue header with the title 'Object Lifetime'. On the left, there is a sidebar with a navigation menu:

- Module 13
- Partha Pratim Das
- Objectives & Outcome
- Constructor Parameterized Overloaded
- Destructor
- Default Constructor
- Object Lifetime
- Automatic Static Dynamic
- Summary

Below the menu is a small video thumbnail of a person speaking.

The main content area contains two bullet points:

- Execution Stages
 - Memory Allocation and Binding
 - Constructor Call and Execution
 - Object Use
 - Destructor Call and Execution
 - Memory De-Allocation and De-Binding
- Object Lifetime
 - Starts with execution of Constructor Body
 - As soon as Initialization ends and control enters Constructor Body
 - Must follow Memory Allocation
 - Ends with execution of Destructor Body
 - As soon as control leaves Destructor Body
 - Must precede Memory De-allocation
 - For Objects of Built-in / Pre-Defined Types
 - No Explicit Constructor / Destructor
 - Lifetime spans from object definition to end of scope

At the bottom of the slide, there is footer text: 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

Just before we move on, this is just to summarize what I have discussed so far in that illustration. That an object may be considered to have five different execution stages because first for the object to exist there has to be a memory, which has to get allocated somehow, either on the stack automatically or in the global common area statically or in the heap dynamically.

So, an allocation is required because without the allocation I do not have a memory

location where to store the different component values. And, as I do the allocation there has to be a binding. That means, there has to be an association between the name of the object and the address that I am using for that object in the memory. So, this is the first execution state that has to happen. And, for all different types of objects this happen in multiple different ways. You need not get worried about this binding aspect right now. As we go over more and more depth of C++, this binding concept will become more and more clear. But, once a binding has been done, that is, I have a memory to create the object, with that memory we will call the constructor implicitly. And as we have just discussed, it executes through the initialization list. And, the object gets constructed, then the execution continues to the constructor body.

The third stage is all different use of the object, till the destructor call happens. And, the destructor again goes through the body, executes to destroy notionally all the different components. And, at the end of the destructor body we will have the destruction process over. And after that, it is time to deallocate the memory. And, once the memory gets deallocated, the debinding happens. That is, the association between the address and the name is no more valid. So, this is what we have mentioned here.

One point to note at this, here in this term of the built-in or predefined types, you may note that notionally all types in C++ are assumed to have constructor or destructor. But, in reality there is no constructor or destructor for the built-in types. They just follow the same syntax for uniformity. But in reality, they are just simple bitwise assignment of values that happens or initialization of values that happen. So for a built-in type, the lifetime basically spans from the point of definition to the end of scope.

(Refer Slide Time: 18:25)



Program 13.14: Complex: Object Lifetime: Automatic

Module 13
Partha Pratim Das
Objectives & Outline
Constructor
Parameterized Constructor
Destructor
Default Constructor
Object Lifetime
Automatic
Static
Dynamic
Summary

```
#include <iostream>
using namespace std;
class Complex { private: double re_, im_;
public:
    Complex(double re = 0.0, double im = 0.0): re_(re), im_(im) // Ctor
    { cout << "Ctor: (" << re_ << ", " << im_ << ")" << endl; }

    ~Complex() // Dtor
    { cout << "Dtor: (" << re_ << ", " << im_ << ")" << endl; }

    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << " (" << re_ << ", " << im_ << ")" << endl; }
};

int main()
{
    Complex c(4.2, 5.3), d(2.4); // Complex::Complex() called == c, then d == objects ready
    c.print(); // Using objects
    d.print();
    return 0;
} // Scope over, objects no more available.
// Complex::~Complex() called == d then c
// Note the reverse order!
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 19

So, this is a more complete example on the lifetime of automatic objects. That is, objects that are necessarily local to a function body or a function parameter and so on. So, this is, these are two objects here. So, if we look into the lifetime, then certainly c will get constructed first. So, the constructor of c will get called first. And, that is why this is what you see at the message statement that when the constructor is called, re and im is set to (4.2, 5.3). So, this is the construction of c.

Subsequently, d is constructed. This is the construction of d, then these are printed. So, that is the use of the objects. And finally, at this point, at the point of return, the destructors will get called. And, I will just minimize this to show you that the destructors have been called and they destroy the two objects.

Now, there is something. One point that you must note very carefully is the destructors are called in the reverse order of construction. That is the default automatic behavior. When the objects are getting constructed, they are getting constructed one after the other. So, c has been constructed first and d has been constructed after that. But, if you look into the destruction, the d is destructed first and c is destructed next. So, you can think of as if the construction, destruction or literally construction, destruction is a LIFO process, where the order in which they are constructed is a reverse order in which they are destructed. As long as you have an automatic or a static object, this construction and destruction rule will have to be followed. So, if say, the object d has a dependence on

object c, then it is logical to construct c and then construct d. But, you will have to remember that d will disappear; have to disappear first as a destruction of d will happen earlier than the destruction of c.

(Refer Slide Time: 20:49)

Module 13	Program 13.15: Complex: Object Lifetime: Automatic: Array of Objects
Partha Pratim Das	<pre>#include <iostream> using namespace std; class Complex { private: double re_, im_; public: Complex(double re = 0.0, double im = 0.0) : re_(re), im_(im) // Ctor { cout << "Ctor: (" << re_ << ", " << im_ << ")" << endl; } ~Complex() // Dtor { cout << "Dtor: (" << re_ << ", " << im_ << ")" << endl; } void opComplex(double i) { re_ += i; im_ += i; } // Some operation with Complex double norm() { return sqrt(re_*re_ + im_*im_); } void print() { cout << "(" << re_ << "," << im_ << ")" << endl; } }; int main() { Complex c[3]; // Default ctor Complex():Complex() called thrice == c[0], c[1], c[2] for (int i = 0; i < 3; ++i) { c[i].opComplex(i); c[i].print(); } // Use array return 0; } // Scope over. Complex::~Complex() called thrice == c[2], c[1], c[0] == reverse order // cleanup</pre>
Object Lifetime	Output: Ctor: (0, 0) Ctor: (0, 0) Ctor: (0, 0) 10+j0i = 0 1i+j1i = 1.41421 2z+j2i = 2.00043 Dtor: (2, 2) Dtor: (1, 1) Dtor: (0, 0)
Automatic Stack Dynamic Summary	NPTEL MOOCs Programming in C++
	Partha Pratim Das

Another example, here we specifically show the construction of array objects. So, we were continuing to use the complex as a class. So, we have an array of three of complex objects that I should mention that if you want to construct arrays of objects, then your class must support a default constructor. The reason is simple. If the constructor is not default, its call will require the parameters to be passed.

Now, if I have an array as in here of `c[3]`, which means the name of the array `c` and I have three objects residing at `c[0]`, `c[1]` and `c[2]`. So, naturally there are three different constructors that have to be called or rather the constructor will have to be called thrice; once for this object `c[0]`, once at this address `c[1]` and then finally at this address `c[2]`. So, there has to be three calls of the constructor. And, this is what you can see here by tracking the message that the constructor prints. And, since it is not convenient notationally to provide the six possible parameters of these three constructors, C++ assumes that if you are constructing array of objects, then the constructor has to be a default one. Then, you are doing something with this array.

So, for just illustration purpose I have defined a member function, `opComplex()`, which does nothing but takes a double value `i` and adds that to both the members. So, kind of it

diagonally shifts the complex number on the complex plane. So, I do that on all the array objects one after the other. For every $c[i]$, for every array object, I applied the complex and do the print. So, these are the different complex numbers that I have now; at the 0th location, I have (0, 0); at the 1st location, I have (1, 1); the 2nd location, I have (2, 2). I have just done this to illustrate the order in which the destructors will happen.

So, when the scope gets over finally at this point of return, then certainly the destructors will have to get called. And by the rule, since this is the order in which the construction happened, naturally the destruction will have to happen in the reverse order. That is, $c[2]$ will destructed first, then $c[1]$, then $c[0]$. A fact that you can see here that the destructor for the complex number (2, 2), happens first, then the destructor for $c[1]$, that is, complex number (1,1) and finally $c[0]$. So, this clearly shows what needs to be done in case of array objects, in terms of their lifetime. So, again from this construction to destruction is a lifetime of all the array elements that we have.

(Refer Slide Time: 23:59)

**Program 13.16: Complex: Object Lifetime:
Static**

Module 13

Partha Pratim Das

Objectives & Outline
Constructor Parameterized Overloaded
Destructor Default Constructor
Object Lifetime Automatic Static Dynamic Summary

```
#include <iostream>
using namespace std;

class Complex { private: double re_, im_;
public:
    Complex(double re = 0.0, double im = 0.0): re_(re), im_(im) // Ctor
    { cout << "Ctor: (" << re_ << ", " << im_ << ")" << endl; }
    ~Complex() // Dtor
    { cout << "Dtor: (" << re_ << ", " << im_ << ")" << endl; }
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "|" << re_ << "+j" << im_ << ")" << endl; }
};

Complex c(4.2, 5.3); // Static (global) object
// Constructed before main starts
// Destructed after main ends

int main() {
    cout << "main() Starts" << endl;
    Complex d(2.4); // Ctor for d
    c.print(); // Use static object
    d.print(); // Use local object
    return 0;
} // Dtor for d
// Dtor for c
```

OUTPUT

```
Ctor: (4.2, 5.3)
main() Starts
Ctor: (2.4, 0)
14.24j5.31 = 6.7624
12.44j0l = 2.4
Dtor: (2.4, 0)
Dtor: (4.2, 5.3)
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 21

This is another example with the same class showing lifetime. Now, here is a main function, but we have defined an object statically here. In the static area is a global static object. So, rest of the class is the same. Here is why you can see the output. And, here the output is critical to be able to understand what is happening. And, please track the constructor output message. So, the constructor output message for (4.2, 5.3), this global static objects you see is printed first, even before main has actually started. So, that is

beautiful information. That is a beautiful understanding.

So far as C was concerned, we always understood that everything starts from main. The computation has to start with main; that is the beginning of. In C++, main, still is an entry point function; main, still is a function that you need to provide in a C++ function. And, that is the first function that will get called by the system. But, the computation necessarily does not start with main. The computation starts with the constructors of static objects which have to be constructed, initialized, before main starts. So, all static objects get constructed before main starts.

So, rest of it is constructing a local object as in here. Then, using both these objects the lifetime continues. And, since d is a local object, when the main reaches the end of scope, the destructor for d gets called. And, what is not visible here is this is the point when the main has actually returned. And, the destructor for this static object c happens, after main returns, which is matching, befitting with the LIFO strategy that the order of destruction has to be reverse of the order of the construction.

So, if I just talk about the simple example, where there is only one static object, the order will construct that static object, call main. Main will construct local objects; use the static as well as local objects, till the scope of main ends. And, at the end of the scope of main, destroy the local objects. The main returns then destroy the static object that was constructed. And, this is; so, actually there is scope for executing code in C++, before main starts and after main has completed.

(Refer Slide Time: 26:54)

The screenshot shows a slide titled "Program 13.17: Complex: Object Lifetime: Dynamic". On the left, there's a sidebar with navigation links: Module 13, Partha Pratim Das, Objectives & Outline, Constructor, Destructor, Default Constructor, Object Lifetime, Automatic, Static, Dynamic, Summary. The main area contains C++ code:

```
#include <iostream>
using namespace std;
class Complex { private: double re_, im_; public:
    Complex(double re = 0.0, double im = 0.0) : re_(re), im_(im) {} // Ctor
    ~Complex() {} // Dtor
    friend ostream << Complex; //友元操作符
    friend istream >> Complex;
    double norm() const { return sqrt(re_*re_ + im_*im_); }
    void print() const { cout << " (" << re_ << " , " << im_ << " )"; }
};
int main() {
    unsigned char buf[100]; // Buffer for placement of objects
    Complex* pc = new Complex(4.2, 5.3); // operator new: allocates memory, calls Ctor
    Complex* pd = new Complex[2]; // operator new []: allocates memory,
                                 // calls default Ctor twice
    Complex* pe = new (buf) Complex(2.6, 3.9); // operator placement new: only calls Ctor
                                                // no allocation of memory, uses buf
    // Use objects
    pc->print();
    pd[0].print(); pd[1].print();
    pe->print();
    // Balance of objects - can be done in any order
    delete pc; // delete: calls Dtor, releases memory
    delete [] pd; // delete[]: calls 2 Dtors, releases mem
    pe->~Complex(); // No delete: explicit call to Dtor
                     // Use with extreme care
    return 0;
}
```

On the right, the output window shows the results of the program execution:

----- OUTPUT -----

```
Ctor: (4.2, 5.3)
Ctor: (0, 0)
Ctor: (0, 0)
Ctor: (2.6, 3.9)
(4.2+5.3j) = 6.7624
(0+0j) = 0
(0+0j) = 0
(2.6+3.9j) = 4.68722
Dtor: (4.2, 5.3)
Dtor: (0, 0)
Dtor: (0, 0)
Dtor: (2.6, 3.9)
```

The last example is in terms of a dynamic allocation. We have seen how to; the different dynamic allocation operators, operator new and so on. So, we again use complex to illustrate that. This is using operator new. I can create a complex object. I can use; create operator new array form to create an array of complex objects. I can do a operator placement, I can do a new placement into a given buffer for an object.

Now, certainly when new happens, then first the new will allocate memory and then the construction will happen. So, like in cases of automatic or static objects, the construction was implicit. Similarly, in new also the construction is implicit. That is the only difference being that before the construction, the new actually will allocate the memory dynamically. So, both these forms of new and array new, will allocate memory and then will call the corresponding necessary constructor.

So, when these are released, like I do delete pc, that is, I am trying to release this object. Then, this delete will call the destructor, corresponding to the call of the constructor and then it will release the memory. So, new and delete are not; we had said earlier that new and delete are like malloc and free. But, they are not exactly malloc and free because malloc only allocates memory, but new allocates memory and also calls the constructor, free only deallocates memory, but delete, calls the destructor and then deallocates memory. So, this difference has to be kept in mind. The total whole output of this program is shown here. You can; you should carefully trace to understand what is going

on. Here, since, it is dynamic allocation, so the user has full freedom as to when to allocate; when to create the object by new and when to destroy it by delete. And, the lifetime is limited between them. Only exception that you will have to remember is if you are doing a placement new, as we had explained earlier, the memory is not to be allocated. It is coming from, provided by the user as in case of buffer. Therefore, you cannot do a delete on a pointer that has been created by placement new.

So, for this object `pe`, which is created by placement new, you will have to do something which is explicit destruction. That is, on that pointer you will have to actually call the destructor of complex. So, this is one of the very rare cases, where you explicitly call the destructor because here it cannot be packaged within the delete operation because delete will need to release the memory, which you do not have here. And, it cannot be implicit because it is a dynamic process, which you want to manage yourself.

(Refer Slide Time: 30:05)

The slide is titled "Module Summary" and features a sidebar with navigation links for Module 13, Partha Pratim Das, Objectives & Outline, Constructor, Parameterized Overloaded, Destructor, Default Constructor, Object Lifetimes, Automatic Storage Duration, and Summary. A small profile picture of Partha Pratim Das is also present. The main content area contains a bulleted list of nine points summarizing object lifetime and constructor/destructor behavior.

- Objects are initialized by Constructors
- Constructors can be Parameterized and can be Overloaded
- Default Constructor does not take any parameter. It is necessary for defining arrays of objects
- Objects are cleaned-up by Destructors. Destructor for a class is unique
- Compiler provides free Default Constructor and Destructor, if not provides by the program
- Objects have a well-defined lifetime spanning from execution of the beginning of the body of a constructor to the execution till the end of the body of the destructor
- Memory for an object must be available before its construction and can be released only after its destruction

NPTEL MOOCs Programming in C++ Partha Pratim Das 23

So, this is the whole process of object lifetime. Please, go through this carefully couple of times because construction, destruction and associated object lifetime is all of the very, one of the core concepts of object management in C++. And, whole of what we will do in the subsequent modules in the subsequent weeks, will critically depend on your understanding of the construction and the destruction process and the validity of the lifetime. All that we have that we have discussed in this module are given in this summary. So, you could just check back that you have understood all these points.

And, thank you very much.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

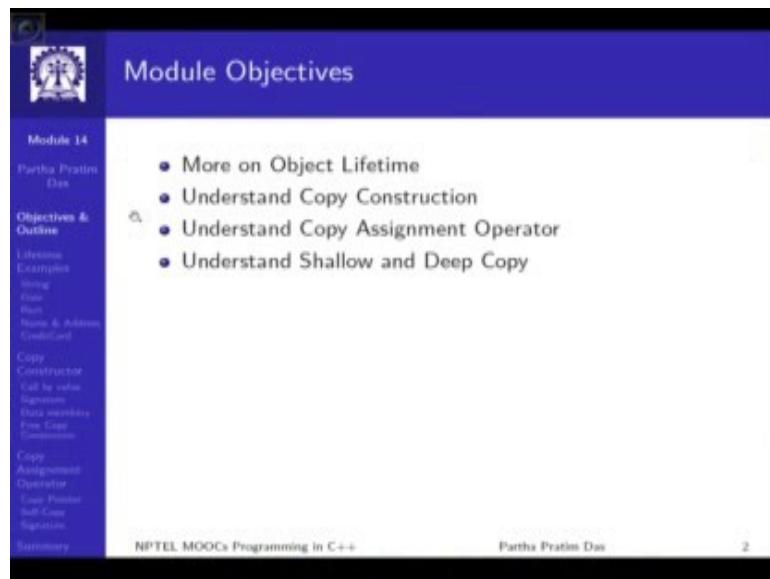
Lecture – 26
Copy Constructor and Copy Assignment Operator

Welcome to module 14 of Programming in C++. In the last module, we have discussed about the construction and destruction of objects and combined with that, we have taken a look in to the lifetime of objects.

We have seen after the construction the object can be used and the construction process actually ends and the object is considered fully constructed, when the initialization list of the object has been completely executed, and the constructor body is about to get executed. Then the object can be used for as long as is required and when the destruction or the destructor call has happened, then the destruction process starts, but the actual destruction happens only when the body of the destructor has completed.

We have seen this with couple of examples and we have also seen the construction and destruction-based object lifetime for automatic, static and dynamically allocated objects. Before we move on to more of object construction in terms of copy construction and copy assignment operation. Will take a little bit more look in to the object lifetime.

(Refer Slide Time: 01:42)



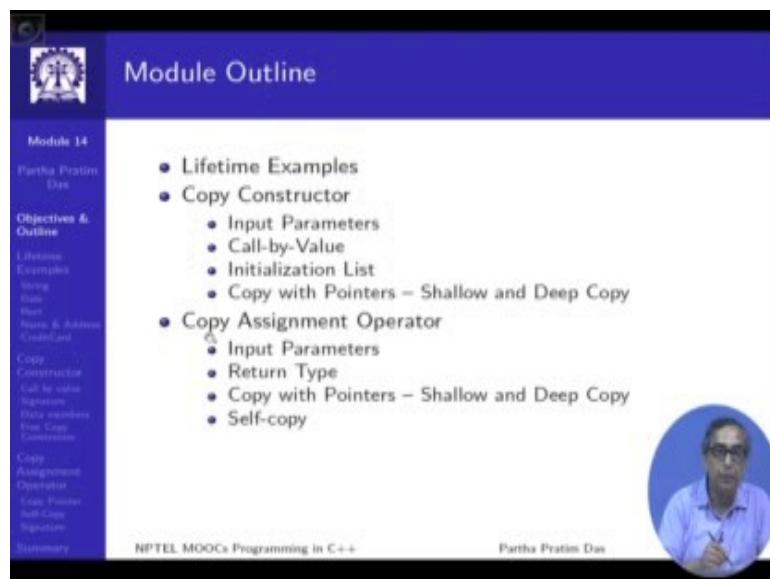
This slide is titled "Module Objectives". It features a sidebar on the left with a navigation menu for "Module 14" by Partha Pratim Das. The menu includes links for "Objectives & Outline", "Lifetime Examples", "Copy Constructor", and "Copy Assignment Operator". The main content area displays a bulleted list of objectives:

- More on Object Lifetime
- Understand Copy Construction
- Understand Copy Assignment Operator
- Understand Shallow and Deep Copy

The footer of the slide includes the text "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and the number "2".

And then we will primarily focus in this module to discuss on how objects can be copied. The process of copy, which in simple terms is, if x is a variable and y is another variable then assigning y to x is making a copy of y into x. Such copies have a lot more of meaning in terms of C++. So, in this module we will primarily take a look in to that and specifically discuss notions of shallow and deep copy.

(Refer Slide Time: 02:29)



This slide is titled "Module Outline". It features a sidebar on the left with a navigation menu for "Module 14" by Partha Pratim Das. The menu includes links for "Objectives & Outline", "Lifetime Examples", "Copy Constructor", and "Copy Assignment Operator". The main content area displays a bulleted list of topics:

- Lifetime Examples
- Copy Constructor
 - Input Parameters
 - Call-by-Value
 - Initialization List
 - Copy with Pointers – Shallow and Deep Copy
- Copy Assignment Operator
 - Input Parameters
 - Return Type
 - Copy with Pointers – Shallow and Deep Copy
 - Self-copy

A circular video player in the bottom right corner shows a video of Partha Pratim Das. The footer of the slide includes the text "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and a small video icon.

Consequently, this is what will be our outline, which is as you know can be visible on the left-hand side of the border.

(Refer Slide Time: 02:43)

```
#include <iostream>
using namespace std;

int init_m1(int m) { // Func. to init m1
    cout << "Init m1: " << m << endl;
    return m;
}

int init_m2(int m) { // Func. to init m2
    cout << "Init m2: " << m << endl;
    return m;
}

class X {
    int m1; // Initialize 1st
    int m2; // Initialize 2nd
public:
    X(int m1, int m2) :
        m1_(init_m1(m1)), // Called 1st
        m2_(init_m2(m2)) // Called 2nd
    { cout << "Ctor: " << endl; }
    ~X() { cout << "Dtor: " << endl; }
}
int main() { X x(2, 3); return 0; }

Init m1: 2
Init m2: 3
Ctor:
Dtor:
```



```
#include <iostream>
using namespace std;

int init_m1(int m) { // Func. to init m1
    cout << "Init m1: " << m << endl;
    return m;
}

int init_m2(int m) { // Func. to init m2
    cout << "Init m2: " << m << endl;
    return m;
}

class X {
    int m2; // Order of data members swapped
    int m1;
public:
    X(int m1, int m2) :
        m2_(init_m2(m2)), // Called 2nd
        m1_(init_m1(m1)) // Called 1st
    { cout << "Ctor: " << endl; }
    ~X() { cout << "Dtor: " << endl; }
}
int main() { X x(2, 3); return 0; }

Init m2: 3
Init m1: 2
Ctor:
Dtor:
```

* Order of initialization does not depend on the order in the initialization list. It depends on the order of data members in the definition.

NPTEL MOOCs Programming in C++

Partha Pratim Das

So, first let us take re-look into the object lifetime. Now, what we want to highlight in terms of object lifetime here are few things, one we want to see that when an object has a number of data members, what is the order in which these data members are constructed or initialized and what is the order in which they are destructed or de-initialized. And when these data members particularly are of user defined types, that they are objects of some other class then what happens to the lifetime of the object, which has the data members and the lifetime of the data member themselves.

So, with that we will start with an example program, which you can see here, it is not a very meaningful class. It is just a class X and it has two data members they are integer data members and they are being initialize certain values in the initialization list.

Now, we are interested to find out, what is the order in which they will get initialized. So, we have done a small trick, while initializing the value instead of just directly putting the value, that is what you actually want to do is take m1 and assign it to this data member instead of doing that we have written a wrapper function, which takes m and returns m.

So, it basically it is input and output are the same, but because this function will get called when this initialization will happen, we will get to know the order in which these functions are getting called or the order in which the initialization happens. So, if you look into this, then you can get to see that the order is first m1 is initialized and then m2 is initialized which is what we would expect to happen.

Now, let us look into the right-hand side, we have the same set of wrapper functions we have the same initialization. The only difference is in the class, we have the same data members, but we have just swap the order of the data members and now you can see between the order of initialization earlier and the order of initialization this time the order of initialization has changed, that is when m1 was the first data member followed by m2 they were initialized in that order. Now, when we have changed this order of m1 and m2, the initialization order also has changed.

From this, we take a very important lesson in terms of the order of initialization of data members in a class; the lesson is the order depends on the order of data members as they are listed in the class. The order in which they are listed and not the order in which they are actually invoked in the initialization list and we will soon show an example to see what consequence this particular property can have.

(Refer Slide Time: 06:09)

```

Module 14
Partha Pratim Das
Objectives & Outline
Lifetime Examples
String
Data Structures
Name & Address
CodeCard
Copy - Constructor
Call by value
Signature
Data members
For Copy Construction
Copy - Assignment Operator
Copy Pointer
Non-Copy Signature
Summary

```

C Style

```

#include <iostream>
using namespace std;

struct String {
    char *str_; // Container
    size_t len_; // Length
};

void print(const String s) {
    cout << s.str_ << endl;
    cout << s.len_ << endl;
}

int main() {
    String s;
    // Init data members
    s.str_ = strdup("Partha");
    s.len_ = strlen(s.str_);
    print(s);
    return 0;
}

```

Partha: 6

C++ Style

```

#include <iostream>
using namespace std;

class String {
    char *str_; // Container
    size_t len_; // Length
public:
    String(char *s) : str_( strdup(s)), len_(strlen(str_)) {
        cout << "ctor: " << print();
    }
    ~String() {
        cout << "dtor: " << print();
        free(str_);
    }
    void print() { cout << "(" << str_ << ")";
    cout << len_ << ")" << endl;
    }
    size_t len() { return len_; }
};

int main() {
    String s = "Partha"; // Ctor called
    s.print();
    return 0;
}

```

ctor: (Partha: 6)
 (Partha: 6)
 dtor: (Partha: 6)

* Note the order of initialization between str_ and len_. What if we swap them?

So, let us go to the next slide. Here, we have a simple string class as you have seen earlier also there is a container which basically is a pointer to character which is suppose to keep as history and another data member gives the length. So, if I initialize it in the C style, then we will initialize it with certain initial value of; here we have used a name Partha. So, we will string duplicate Partha {strdup("Partha")} into the str field and keep its length in the len field and then we can print. So, it will print whatever is the string and its length.

When we write this in C++ in terms of the class, then certainly we will have the same data members, but we write a constructor and in the constructor initialization, the initialization list, we string dup the given string s first and then we take the str field that has been created, that is the string that has been duplicated, we go and compute the length of that to set the len field and certainly we get the similar set of results when we execute them. So, this is the example we are talking of. Now, the question is, what if between these two data members if you swap them, will this program will work or will this program have some difficulties. So, let us just try to swap and this is the example that you can see.

(Refer Slide Time: 07:56)

Program 14.04: A Simple String Class – Fails for wrong order of data members

```
#include <iostream>
using namespace std;

Module 14
Partha Pratim Das
Objectives & Outcome
Lecture Examples
String Data Structures
Name & Address Credential
Copy Constructor Call by value Assignment Operator Overloading Free Copy Constructor
Copy Assignment Operator Copy Pointer Call by Reference Summary
```

Microsoft Visual Studio

Unhandled exception at 0x00A4E000 (msvcr100.dll) in Module 14.exe: Microsoft C++ exception: `std::bad_alloc` occurred in `main()`.

Break when this exception type is thrown Open exception settings

Partha Pratim Das

The slide shows a screenshot of Microsoft Visual Studio. The title bar says "Microsoft Visual Studio". A message box is displayed with the text: "Unhandled exception at 0x00A4E000 (msvcr100.dll) in Module 14.exe: Microsoft C++ exception: std::bad_alloc occurred in main()". There is a checkbox labeled "Break when this exception type is thrown" and a link "Open exception settings". Below the message box is a portrait of Partha Pratim Das.

The only difference from the earlier one is, we have swapped the order of these two data members. The constructor and the initialization list has not been changed, they are exactly the same, but just the order of the data members has been swapped and as you can see here, in this I have executed this in Microsoft Visual Studio, we can see that the program crashes.

The reason is understandable because as I said earlier that the order of initialization depends on the order of the data members on the list. So, since len here is given earlier, this particular initialization happens first and when this happens, the str field. This str field has not yet been initialized no memory has been allocated in this pointer and no string has been copied. So, the call to str len actually get what we commonly say garbage values and therefore, from that we get this kind of an error.

So, len will produce this is the points are highlighted here for your reference and because of this call to strlen with uninitialized values, we will get this kind of an error. So, this is just to highlight through a simple example that the order of the data member can be critical and while writing the initialization list, you should be aware of the order in which the data members are actually created.

(Refer Slide Time: 09:42)



```
#include <iostream>
using namespace std;

char monthNames[12] = { "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                      "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };
char dayNames[10] = { "Monday", "Tuesday", "Wednesday", "Thursday",
                     "Friday", "Saturday", "Sunday" };

class Date {
    enum Month { Jan = 1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec };
    enum Day { Mon, Tue, Wed, Thr, Fri, Sat, Sun };
    typedef unsigned int UINT;
    UINT date; // Month month.; UINT year.;

public:
    Date(UINT d, UINT m, UINT y) : date(d), month_(Month(m)), year_(y)
    { cout << "ctor: "; print(); }
    ~Date() { cout << "dtor: "; print(); }
    void print() { cout << date_ << "/" << monthNames[month_-1] << "/" << year_ << endl; }
    bool validate() { /* Check validity */ return true; } // Not implemented
    Day day() { /* Compute day from date using time.h */ return Mon; } // Not implemented
};

int main() {
    Date d(30, 7, 1961);
    d.print();
    return 0;
}

ctor: 30/Jul/1961
~Date: 30/Jul/1961
dtor: 30/Jul/1961
```

Now, let us look at few more simple examples. These are just initially for illustration and then we will put this together into building, a little bit of a bigger class so that we can see the order of lifetime or the order of construction destruction better. So, here we just show a simple Date class. So, this Date class has 3 data members; date, month and year. So, which are basically either numbers or they are enumerated types like Jan, Feb and so on.

So, which are basically sub types of integer we use them to initialize a date in the constructor and therefore, if you execute this program you will get this kind of an output where first the date will be constructed as in here, then it will be printed using this print and finally, it will get destructed. There is nothing more in this example will just use more of these examples subsequently.

(Refer Slide Time: 10:49)

```

Module 14
Partha Pratim Das
Objectives & Outline
Lifetime Examples
String
Date
Rect
Point & Address
FinalCard
Copy Constructor
Call by value
Signatures
Data members
For Copy Construction
Copy Assignment Operator
From Pointer
Self-Copy Signatures
Summary
NPTEL MOOCs Programming in C++

```

Program 14.06: Point and Rect Classes: Lifetime of Data Members or Embedded Objects

```

#include <iostream>
using namespace std;

class Point {
    int x;
    int y;
public:
    Point(int x, int y) {
        x = x; y = y;
        cout << "Point ctor: ";
        print(); cout << endl;
    }
    ~Point() { cout << "Point dtor: ";
        print(); cout << endl; }
    void print() {
        cout << "(" << x << ", " << y << ")";
    }
};

int main() {
    Rect r(0, 2, 6, 7);
    cout << endl; r.print(); cout << endl;
    cout << endl;
    return 0;
}

class Rect {
    Point TL;
    Point BR;
public:
    Rect(int tlx, int tly, int brx, int bry) {
        TL.tlx = tlx; TL.tly = tly;
        BR.brx = brx; BR.bry = bry;
        cout << "Rect ctor: ";
        print(); cout << endl;
    }
    ~Rect() { cout << "Rect dtor: ";
        print(); cout << endl; }
    void print() {
        cout << "(" << TL.x << ", " << TL.y << ") ";
        cout << "(" << BR.x << ", " << BR.y << ")";
    }
};

Point ctor: (0, 2)
Point dtor: (0, 2)
Rect ctor: ((0, 2), (6, 7))
((0, 2), (6, 7))
Rect dtor: ((0, 2), (6, 7))
Point dtor: (0, 2)

```

Partha Pratim Das

Now, I show another example for the lifetime, which is Point and Rectangle classes, so a point class has two data members the x and y coordinates of a point. It has constructors which initialize these points and a rectangle is basically a pair of points, where TL_ stands for top left and BR_ stands for bottom right, that is a two corner points of a rectangle, if I specify that, then the rectangle is fully specified.

Now, if I want to initialize if you look carefully then I need actually initialize these two. So, for rectangle we have a constructor, which specifies four integers, of that the first two are meant for the x and y coordinate of the top left point the next two are x and y coordinates of the bottom right point. So, we take the first two and construct the TL_ component take the next two to construct the BR_ component and the rectangle gets constructed fully.

So, if we look at if you try to see the order in which the constructors get called, we will see that the first this constructor gets called. Why, because these are the two integers which is tlx and tly, TL_ is the first data member of Rect class. So, naturally this is the first initialization to happen in the initialization list. So, this initialization means that a point has to get constructed, which means that for this initialization this particular constructor will have to get called, this constructor gets called. Accordingly, this particular body of the constructor prints this output, it says that a point with (0, 2) has been constructed.

Next, the next two integers are taken; the second element in the initialization list is to be constructed. Again, another point is constructed to the same constructor and you get the output for that; then there is a print which shows that the rectangle, I am sorry. Then once that has been done, then the body of the rectangle constructor gets executed. So, you get this output which says that a rectangle with TL_ (0, 2) and BR_ (5, 7) has been constructed, and then you print. So, the print comes in here and finally, the destruction process starts, when the destruction process starts at this point, we have already explained that this Rect is an automatic object. So, the destruction will start, when the scope of that automatic object ends, which is the closing bracket of the block of main. So, what will happen at this? This was the construction order. So, the destruction order will be exactly the reverse that is a LIFO.

So, the first call will happen to the destructor of the rectangle. So, first this gets executed and when it comes to the end of the destructor body of rectangle, at that time the rectangle is ready to be destroyed, which means that now the data members of the rectangle object has to be destroyed. So, the next one which was constructed just before this that is the BR_, this is basically for BR_ that will get destructed. So, for that the

destructor of the BR_, the destructor of BR_ will get called and you get this output and finally, the first object that was constructed which is the object TL_, the destructor for TL_ will get called and that results in the results in the messages that we have.

Now, it is kind of a one function destructor function calling the next and once that completes then it calls the next destructor function and so on and when all these are over then the destruction process is over. So, this is just too clearly highlighted to you, as to what happens when you have data members which are of user define types that is data members, which are not just built-in the data members, which actually have their constructors and destructors. So, you will have to remember that they are constructed in the order in which they are listed in the class and they are destructed in the reverse order of their construction.

(Refer Slide Time: 15:39)

```

Module 14
Partha Pratim Das
Objectives & Outline
Lifetime Examples
Using Base Class
Name & Address
Copy Constructor
Call by value
Signatures
Data members
File Copy Constructor
Copy Assignment Operator
Copy Pointer
Self-Copy Signatures
Summary

Program 14.07: Name & Address Classes



```

#include <iostream>
using namespace std;

#include "String.h"
#include "Date.h"

class Name { String firstName_, lastName_; public:
 Name(const char* fn, const char* ln) : firstName_(fn), lastName_(ln)
 { cout << "Name ctor: " ; print(); cout << endl; }
 ~Name() { cout << "Name dtor: " ; print(); cout << endl; }
 void print()
 { firstName_.print(); cout << " " ; lastName_.print(); }

};

class Address {
 unsigned int houseNo_;
 String street_, city_, pin_;
public:
 Address(unsigned int hn, const char* sn, const char* cn, const char* pn)
 { houseNo_<(hn); street_<(sn); city_<(cn); pin_<(pn);
 cout << "Address ctor: " ; print(); cout << endl; }
 ~Address() { cout << "Address dtor: " ; print(); cout << endl; }
 void print()
 { cout << houseNo_ << " "
 street_.print(); cout << " "
 city_.print(); cout << " "
 pin_.print();
 }

};


```


NPTEL MOOCs Programming in C++
Partha Pratim Das

```

Let us move on, in the next couple of slides we will just build up to show you a bigger example. So, in this slide we show two more classes, one is a Name class designed to keep or maintain names of a person. So, it has two different data members; firstName_ and lastName_. They are of the type string that we have already discussed. So, we have included the string class as in the header. So, for these two data members we provide the parameters to the constructor and they get constructed in the initialization list.

We have another class Address which is to keep a house number (houseNo) and three strings; street_, city_ and pin_, which are strings describing the respective data members. So, they will be constructed in the constructor of address and will be printed subsequently. So, this is what using the string, now we have a name class which can maintain names of persons and address class which can maintain address of persons and so on. We are just building this up to for a bigger example.

(Refer Slide Time: 16:59)

```

Module 14
Partha Pratim Das
Objectives & Outcome
Lecture Examples
String
Data Structures
Matrix & Address
CreditCard
Copy Constructor
Call by value
Signature
Reference
Pointers
Function Overloading
Copy Assignment Operator
Local Function
Self-Calls
Registers
Summary

Program 14.07: CreditCard Class

class CreditCard { typedef unsigned int UIINT;
    char cardNumber_[17]; // 16-digit (character) card number as C-string
    Name holder_;
    Address addr_;
    Date issueDate_, expiryDate_;
    UIINT cvv_;
public:
    CreditCard(const char* cNumber, const char* fn, const char* ln,
               unsigned int bn, const char* sn, const char* cn, const char* pin,
               UIINT issueMonth, UIINT issueYear, UIINT expiryMonth, UIINT expiryYear, UIINT cvv) :
        holder_(fn, ln), addr_(sn, cn, pin),
        issueDate_(bn, issueMonth, issueYear),
        expiryDate_(ln, expiryMonth, expiryYear), cvv_(cvv)
    {
        strcpy(cardNumber_, cNumber); cout << "CC num: "; print(); cout << endl;
    }
    void print()
    {
        cout << cardNumber_ << endl;
        holder_.print(); cout << endl;
        addr_.print(); cout << endl;
        issueDate_.print(); cout << endl;
        expiryDate_.print(); cout << endl;
        cout << cvv;
    }
};

int main()
{
    CreditCard cc("5331711934640927", "Sherlock", "Holmes",
                 "221, Baker Street", "London", "NW 1XE", 7, 2014, 19, 2015);
    cout << endl; cc.print(); cout << endl << endl;
    return 0;
}

```

NPTEL MOOCs Programming in C++
Partha Pratim Das

Now, this is the example which we were interested in; let consider a class to represent a credit card. I am sure all of you have or at least have known, how a credit card can be described. A credit card has a card number which is typically a 16-character string. It certainly has embossed in it, the name of the holder, the address of the holder is given though is not written on the card. A card has an issue date, it has an expiry date and on the reverse of the card, we have a verification number called the cvv number. So, if you want to describe a credit card object, then this typically; this is a very minimal description that we can have for a credit card object.

So, for at the constructor which is here a big list, this is actually the prototype of the constructor. This is three lines which give us the string giving the credit card number then it gives the names; first and second name of the holder. Then it gives the house

number, street name, city, and pin for the address of the holder. Then it gives the issue date, that is issue and expiry dates. These use the date class that we have defined and finally, the cvv and we use the initialization list to initialize.

Therefore, if we use this class to create a credit card object for say, Mister Sherlock Holmes, staying in 221 Baker street, London, then the invocation of the constructor will look something like this and with that let us take a look into how the construction-destruction will, how will the lifetime look like.

(Refer Slide Time: 19:01)

The screenshot shows a programming interface with a sidebar on the left containing navigation links such as 'Module 14', 'Partha Pratim Das', 'Objectives & Outline', 'Lifetime Examples', 'String', 'Date', 'Pair', 'Name & Address', 'CreditCard', 'Copy', 'Constructor', 'Call by value', 'Signature', 'Data members', 'Free Copy Construction', 'Copy Assignment Operator', 'Copy Pointer', 'Self Copy Signature', and 'Summary'. The main area has a title 'Program 14.07: CreditCard Class: Lifetime Chart'. It displays two sections: 'Construction of Objects' and 'Destruction of Objects'. The 'Construction of Objects' section shows the creation of a CreditCard object with the following details:

```

Module 14
Partha Pratim Das
Objectives & Outline
Lifetime Examples
String Date
Pair
Name & Address
CreditCard
Copy
Constructor
Call by value
Signature
Data members
Free Copy Construction
Copy Assignment Operator
Copy Pointer
Self Copy Signature
Summary

Program 14.07: CreditCard Class: Lifetime Chart

Construction of Objects
    String: Sherlock
    String: Holmes
    Name: Sherlock Holmes
    String: Baker Street
    String: London
    String: NW1 6EE
    Address: 221 Baker Street London NW1 6EE
    Date: 1/Jul/2014
    Date: 1/Dec/2016
    CC: 5321711934640027 Sherlock Holmes 221 Baker Street London NW1 6EE 1/Jul/2014 1/Dec/2016 811

    class CreditCard { char cardNumber_[17];
        Name holder_;
        Address addr_;
        Date issueDate_, expiryDate_;
        UInt cvv_, };
        class Name { String firstName_, lastName_; }
        class Address { unsigned int houseNo_;
            String street_, city_, pin_; }
        class Date { enum Month;
            UInt day_, Month month_, UInt year_, }; }

    5321711934640027 Sherlock Holmes 221 Baker Street London NW1 6EE 1/Jul/2014 1/Dec/2016 811

```

The 'Destruction of Objects' section shows the destruction of the same CreditCard object:

```

Destruction of Objects
    CC: 5321711934640027 Sherlock Holmes 221 Baker Street London NW1 6EE 1/Jul/2014 1/Dec/2016 811
    ~CreditCard()
    ~Name()
    ~Address()
    ~Date()
    ~String()
    ~UInt()

    5321711934640027 Sherlock Holmes 221 Baker Street London NW1 6EE 1/Jul/2014 1/Dec/2016 811

```

At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

So, this is how the object is getting constructed and just for reference, I have put the definition of the credit class and other classes here, this is just for your reference, but this is the output, this whole thing is output. So, this is the construction phase, where naturally if you want to construct, the first field here, is a card number which is nothing, but a string. So, there is no constructed to be explicitly called. So, therefore, there is no output, but the next certainly what will get constructed is the holder.

So, the constructor of name will get called. So, for that then constructor of the name has two fields, the first name and the second name which are strings. So, first thing that gets called is the first name string then the second name string and then the construction of

the name object. Then address starts, again the house number is an integer. So, that has no constructor, but the other three strings will get constructed next, which give you the street name, the city name and the pin and then the address object gets constructed. Then subsequently the two data objects get constructed here and here and finally, the credit card object get constructed.

So, this is just shows and I have intentionally done this kind of indentation to show that the more indented is a statement is what has been constructed earlier. So, this is the order in which this will get constructed. So, these two will construct this, these two will finally give these. These are the separate objects and all of these together will finally construct the credit card object. Then the credit card can be used which we just show by print and then the destruction order if you look at and I will leave this as for your study to look at carefully the order of the destruction is exactly in the reverse order of.

So, if you just read this list, bottom to top that is the order in which the destruction will happen for this is the objects and that is just for your practice also, I would suggest it is naturally in my quick description you may not be able to see all of the details here, but the whole program is given in the presentation and you have or if go through the presentation and I would suggest that you also try to run it in your system and try to see that you are getting the same result and get convinced about what is the different lifetime of objects that you get when you have nested objects this data member like this.

(Refer Slide Time: 22:06)

The slide has a blue header bar with the title 'Copy Constructor'. On the left, there is a sidebar with a logo and a navigation menu. The main content area contains three bullet points under the heading 'We know:'.

- We know:
Complex c1 = {4.2, 5.9}; // or c1(4.2, 5.9)
invokes
Constructor Complex::Complex(double, double);
- Which constructor is invoked for?
Complex c2(c1);
Or for?
Complex c2 = c1;
- It is the **Copy Constructor** that takes an object of the same type and constructs a copy.
Complex::Complex(const Complex &);

At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

Now, we will move into discussing a new kind of constructor and the reason I discussed object lifetime here, is we will see that with this new kind of constructor the lifetime of objects will get new dimensions to understand. So, we just start by showing a simple example say, we know complex can be constructed in this way and that will call a complex constructor like this we have seen the complex class quite often, but suppose I write it like this what does that mean.

So, just see the difference, here it is written like this, here it is or if I write it like this. There is a main difference is here, I have specified the parameters of the constructor the two double numbers one after the other by comma. Whereas, here I have used a complex object itself to initialize another complex object c2.

When I try to do this, then I am doing a construction which is a special kind of construction which is known as the copy construction and a copy constructor looks like this, this is just a constructor. So, it is Complex::Complex, only difference being that it takes another complex object as a parameter and it takes it as a reference parameter and we are using const in front of this, let us slowly understand why we do all these.

(Refer Slide Time: 22:53)



Program 14.08: Complex: Copy Constructor

Module 14
Partha Pratim Das
Objectives & Outcome
Lecture Examples
String Date
Matrix & Address
CreditCard
Copy Constructor
Call by value
Assignment
Overloading
Copy Constructor
Copy Assignment
Destructor
Exception Handling
Scope Resolution
Summary

```
#include <iostream>
#include <cmath>
using namespace std;
class Complex { double re_, im_;
public:
    Complex(double re, double im) : re_(re), im_(im) // Constructor
    { cout << "Complex ctor: "; print(); }
    Complex(const Complex& c) : re_(c.re_), im_(c.im_) // Copy constructor
    { cout << "Complex copy ctor: "; print(); }
    ~Complex() { cout << "Complex dtor: "; print(); }
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "(" << re_ << im_ << ") = " << norm() << endl; }
};

int main() {
    Complex c1(4, 2, 5.3); // Constructor = Complex(double, double)
    c2=c1; // copy Constructor = Complex(const Complex&)
    c3 = c2; // Copy Constructor = Complex(const Complex&)

    c1.print(); c2.print(); c3.print();
    return 0;
}

//output
Complex ctor: (4.2+5.3i) = 6.7624 // Char: c1
Complex copy ctor: (4.2+5.3i) = 6.7624 // Char: c2 of c1
Complex copy ctor: (4.2+5.3i) = 6.7624 // Char: c3 of c2
(4.2+5.3i) = 6.7624 // c1
(4.2+5.3i) = 6.7624 // c2
(4.2+5.3i) = 6.7624 // c3
Complex dtor: (4.2+5.3i) = 6.7624 // Dtor: c1
Complex dtor: (4.2+5.3i) = 6.7624 // Dtor: c2
Complex dtor: (4.2+5.3i) = 6.7624 // Dtor: c3
```

NITEL MOOCs Programming in C++ Partha Pratim Das 13

Let us look into an example here is the total complex class. So, this is the constructor that we have seen earlier. These are copy constructor that is will specifically have to look into this part of the code that it takes a complex object and then it initializes the data members from the data members of the complex objects. So, the object that you want to copy from is c - the re data member, re_ data member is c.re. So, we take that and put it to re. Similarly, we take c.im and put it to im. So, basically what happens is the new object which is getting constructed by this constructor has identical values in re and im fields from the object it is copying from.

So, in this context if you look into the construction of a couple of different objects then naturally the first construction is simply using are parameterized constructor of two doubles. The second one will use the copy constructor because it takes the c1 object which is a complex object and uses the copy constructor to construct c2. Similarly, the next one will take c2 and copy construct to find of to construct c3 as copies of these. So, after this construction if you try to print them as we do it here then we find that all of them have become identical. They have this identical construction values and, in the print, they show identical values and, in the destruction, also they show identical values. So, this is how copies can be made of objects very easily to this copy constructor mechanism.

(Refer Slide Time: 25:56)

The slide has a blue header with the title 'Why do we need Copy Constructor?'. The left sidebar contains a navigation menu with items like 'Module 14', 'Partha Pratim Das', 'Objectives & Outline', 'Lecture Examples', 'String', 'Date', 'File', 'Name & Address', 'CreditCard', 'Copy Constructor' (which is bolded), 'Copy Assignment Operator', 'Exception', 'Final Project', 'Solutions', and 'Summary'. The main content area lists reasons for needing copy constructors:

- Consider the **function call mechanisms** in C++:
 - *Call-by-reference*: Set a reference to the actual parameter as a formal parameter. Both the formal parameter and the actual parameter share the same location (object)
 - *Return-by-reference*: Set a reference to the computed value as a return value. Both the computed value and the return value share the same location (object)
 - *Call-by-value*: Make a copy (clone) of the actual parameter as a formal parameter. This needs a **Copy Constructor**
 - *Return-by-value*: Make a copy (clone) of the computed value as a return value. This needs a **Copy Constructor**
- **Copy Constructor** is needed for **initializing the data members** of a UDT from an existing value

At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

Now, the question certainly is why we need copy constructors and there are primarily two reasons that copy constructors are provided or need to be provided in C++. To understand, first consider the function call mechanism. As we have already seen, when we talked about better C part of C++, we talked about references; we talked about call by reference or return by reference mechanism. So, if you just recap those then there are four things, we can do we can call by reference in which case the formal parameter actually takes the reference to an actual parameter, that is both the formal and the actual parameter share the same memory location, share the same object.

The same thing happens at the time of return, the value that you return by from the function and the value that you get from the calling function. Basically, these two objects are same if you would return by reference whereas, if you do call by value then you need to make a copy of the actual parameter as a formal parameter, you need to make a copy. Please, this is what most significant point is.

Now, as long as the objects that we pass or the values that we pass, they were of the built-in type making this copy was straight forward because it was just making bit copy of an int or a double or a character and so on. But, when we have user defined objects then we actually need, given actual parameters. So, this is an actual parameter which is

say an object c and I need a formal parameter say f. So, this f will have to be an object of the same type and, but it must be different from c because I want to make a copy and the values of the data members of c, somehow needs to be copied to f. So, the purpose of the copy constructor is significantly to achieve this process of call by values.

So, if a user defined type does not have or is not provided with copy construction process if this is not provided with the copy construction then the consequences, the objects of that class, objects of that user define type cannot be passed as call by value mechanism to any function. The similar observations will happen if you want to return a value, return something from a function by value because we will again need copy constructor to copy the value. The other situation where copy constructor is needed is for initializing the data members.

You have already seen that we have been copying one value into another regularly in the previous examples of object lifetime, but the copied values were typically where whatever data members we had for construction they were typically built-in types, but if I have to copy a value of a user define type as a data member then I will again have faced with the same situation as the call by value situation. So, initializing the data members of UDT will also need the existence of a copy constructor without that a data member of that corresponding type cannot be defined.

We have just seen; we have revisited the object lifetime and we have specifically taken a deep look into the different object lifetime scenarios, particularly with user define types and discussed the issue of ordering of data members and their consequence on the object lifetime and we have just got introduced to the copy constructor.

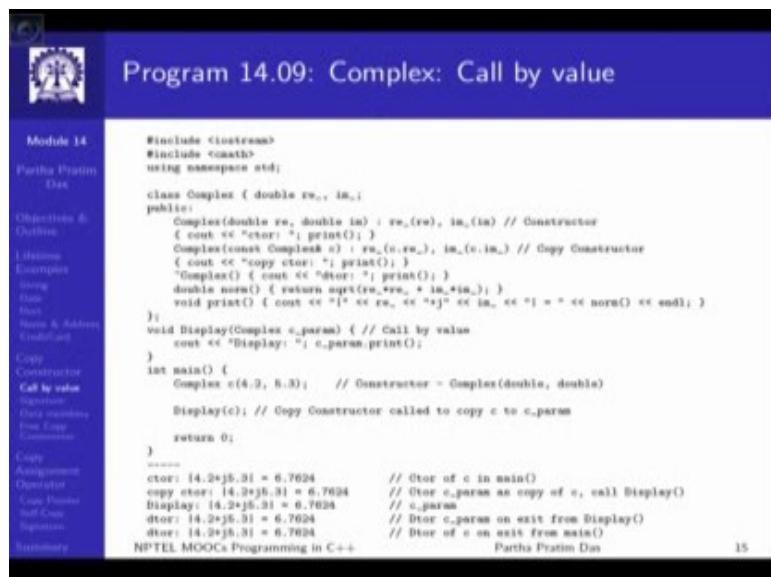
Programming in C ++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 27
Copy Constructor and Copy Assignment Operator (Contd.)

Welcome back to Module 14 of Programming in C++. We have been discussing about copy construction process. We have introduced what a copy constructor is? It is a special constructor which takes an object and makes a copy of it.

So, in that process, it copies each and every data member in whatever way the programmer has decided to do and we have explained that the copy construction is extremely necessary for the purpose of call by value, return by value mechanism and for initializing data members which are part of another object which is being copied. So, if you copy an object, you need to copy its data members. So, you will again copy constructor for that.

(Refer Slide Time: 01:10)



```
#include <iostream>
#include <cmath>
using namespace std;

class Complex { double re_, im_; public:
    Complex(double re, double im) : re_(re), im_(im) {} // Constructor
    Complex(const Complex& c) : re_(c.re_), im_(c.im_) {} // Copy Constructor
    ~Complex() {} cout << "dtor: " << endl;
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "[ " << re_ << " " << im_ << " ] " << endl; }
};

void Display(Complex c_param) { // Call by value
    cout << "Display: " << c_param.print();
}

int main() {
    Complex c(4.2, 6.3); // Constructor = Complex(double, double)
    Display(); // Copy Constructor called to copy c to c_param
    return 0;
}

ctor: 4.2+j6.3 = 6.7624 // Dtor of c in main()
copy ctor: 4.2+j6.3 = 6.7624 // Dtor c_param as copy of c, call Display()
Display: 4.2+j6.3 = 6.7624 // c.param
dtor: 4.2+j6.3 = 6.7624 // Dtor c.param on exit from Display()
dtor: 4.2+j6.3 = 6.7624 // Dtor of c on exit from main()

NPTEL MOOCs Programming in C++ Partha Pratim Das 15
```

So, now we will take quick examples into this. So, again back to our complex class, there is nothing different here, except we have added a function display which takes a complex

number here and prints it and the wave we have designed, this display function is the complex number is passed as a value. So, it is a call by value. So, now if you look into the order in which the constructors are called naturally, first this will get executed. So, the constructor gets invoked which is this constructor; these output.

Then, once c has been constructed, it has to come here which mean that before the function can be called this actual parameters, C has to be copied to this formal parameter c param, and this process is the copy construction process. So, at this point the copy constructor that is this constructor will get invoked and you can see that it is printing the message copy constructor. In this case, it has the same set of member values because that is how it has been set, but it is a different constructor from the original constructor of C. Then, the display happens and that is the function executes. So, it prints. So, this is what you get to see and then, the function is making an exit from this point, there is a control we will have to now come back to here.

When this happens naturally, let me clean up. When the function is ended, the scope of this particular parameter, the formal parameter which is like a function local ends at this point, this also is an automatic object and since it ends at this point, naturally the objects c param that was constructed. So, the c param, this object will no more be available after the display function has returned a control to main.

Therefore, it has to get destructed. So, now, it will get this call of the destructor for c param. Then, the control comes here and then, we are about to return and when I reach the end of the scope of main, the destructor for c gets call and this is how the order goes. So, you can easily see that whenever you will do call by value, you will be able to see such copy construction and the corresponding destructor of the object which is a formal parameter to happen.

(Refer Slide Time: 04:12)

The slide is titled "Signature of Copy Constructors". It includes a sidebar with navigation links for Module 14, Partha Pratim Das, Objectives & Outline, Definitions, Examples, Using Class, Best Practices, Names & Address Considerations, Copy Constructor (Call by value Signature), Copy Assignment Operator, Copy Pointer, Copy Constructor Signature, and Summary. The main content discusses the signature of copy constructors, showing examples like MyClass(const MyClass& other); and MyClass(MyClass& other);. It also notes that MyClass(MyClass* other); is an infinite loop due to call-by-value mechanism.

- Signature of a Copy Constructor can be one of:
 - MyClass(const MyClass& other); // Common
// Source cannot be changed
 - MyClass(MyClass& other); // Occasional
// Source needs to change
 - MyClass(volatile const MyClass& other); // Rare
 - MyClass(volatile MyClass& other); // Rare
- None of the following are copy constructors, though they can copy:
 - MyClass(MyClass* other);
 - MyClass(const MyClass* other);
- Why the parameter to a copy constructor must be passed as Call-by-Reference?
 - MyClass(MyClass other);

The above is an infinite loop as the call to copy constructor itself make copy for the Call-by-Value mechanism.

NPTEL MOOCs Programming in C++ Partha Pratim Das

Let me specifically talk about the signature of copy constructor as C++ has a well specified signature of a copy constructor. This is the signature that is most commonly used. So, I have highlighted it here. This is a common one which takes the object as a call by reference, as a reference object. You can see the reference here. Since it is taken as a reference. As you know, it can be changed because it shares the same memory location as of the actual parameter that we put a const. So, you are saying if you are making copy, then whichever we are coping from that object cannot be changed.

So, that is, what is clearly specified by this signature and that is the most commonly copy constructor that will always see, but occasionally we may drop this const and have a copy constructor like this, where we are saying that while we are coping, it is possible that we also change the object that we had copying from. I am sure this will sound very weird to you right now, but please bear with me. At an appropriate point, I will show examples of why having this features is very important for certain designs and then, there are certain other copy constructors which use volatile data. So, those are specified here.

They are very rarely required specifically if you are using embedded system programming. So, you may look this up when you are doing that kind of a programming and also, you please note that some of the similar signatures like, I could have passed the

object, I want to copy from, I could pass pointer to that object or pointer to the constant object and so on. These C++ do not recognize them as copy constructors. So, if I provide them, then they will be taken as just another constructor, but they will not be invoked at the time of call by value. So, this will have to be known in mind and a final observation is what if I wrote my copy constructor like this and what if I passed the object to copy from as a parameter to the constructor copy, constructor with call by value mechanism and not call by reference.

I can see that this is not going to work because a copy constructor is also a function. So, if we take a parameter other at the time of call by value, then this parameter will also need to be made available to this constructor. So, this itself is a call by value which will mean that this will again call the copy constructor and to call the copy constructor, it needs to do call by value. To do this call by value, it needs to call the constructor. The constructor needs to be called by value and so on so forth. So, it simply keeps on going. This eventually turns out to be an infinite loop and it could not have worked. So, you cannot pass the object to a copy constructor by call by value.

(Refer Slide Time: 07:50).

Module 14:

- Partha Pratim Das
- Objectives & Outcome
- Learning Examples
- Using Data Types
- Name & Address
- Cardinal
- Copy Constructor
- Call by value
- Data members
- Common
- Copy Assignment Operator
- Copy Pointers
- Self-Copy Registration
- Summary

**Program 14.10: Point and Rect Classes:
Default, Copy and Overloaded Constructors**

```
#include <iostream>
using namespace std;
class Point { int x, int y; public:
    Point(int x, int y) : x(x), y(y) { cout << "Point ctor: " << print() << endl; } // Constructor (Ctor)
    Point() : x(0), y(0) { cout << "Point ctor: " << print() << endl; } // Default Constructor (Dctor)
    Point(const Point& p) : x(p.x), y(p.y) { cout << "Point cc: " << print() << endl; } // Copy Constructor (CCtor)
    ~Point() { cout << "Point dtor: " << print() << endl; } // Destructor (Dtor)
    void print() { cout << "(" << x << ", " << y << ")"; }
};

class Rect { Point tl, Point br; public:
    Rect(int tlx, int tly, int bx, int by) { cout << "Rect ctor: " << print() << endl; } // Ctor = Uses Ctor for Point
    Rect(const Point& p1, const Point& p2, int bx, int by) { cout << "Rect cc: " << print() << endl; } // CCtor for Point
    Rect(const Point& p1, int bx, int by) { cout << "Rect cc: " << print() << endl; } // CCtor for Point
    Rect(const Point& p1, int bx, int by) { cout << "Rect cc: " << print() << endl; } // CCtor for Point
    ~Rect() { cout << "Rect dtor: " << print() << endl; } // Dtor
    void print() { cout << "(" << tl.x << ", " << tl.y << ") - (" << br.x << ", " << br.y << ")"; }
};

// When parameter (tlx, tly) is set to TL by TL::(tlx, tly), parameterized Ctor of Point is involved
// When parameter p1 is set to TL, by TL::(p1), Ctor of Point is involved
// When TL is set by default in Dctor of Rect, Dctor of Point is involved
// When member r.TL is set to TL, by TL::(r.TL), CCtor of Rect: CCtor of Point is involved
```

NPTEL MOOCs Programming in C++
Partha Pratim Das
17

Let us look into some bigger examples. These are naturally little bit longish code. So, these are more for you self study than the discussions here. I will just outline what I am

trying to demonstrate here is we are showing default copy and overloaded constructors here. So, there is a constructor, there is a copy constructor, there is a default constructor for the point plus, that we have seen earlier, then we have several constructors for the rectangle class, we have 5 constructors.

The first one takes four points, the second one takes I am sorry, the first one takes four integers that is like why coordinates of the two corner points, the second one takes two different corner points, the third one takes a point and coordinates of the other point and then, last one there is a default one and the last one is a copy constructor.

So, you can try to see, you can try to convince yourself studying all of these constructors and their initialization list as to which constructors will get called. For example, I would just focus on this, the constructor which takes two points. Now, naturally that constructed needs to take two point T 1 and copy it to the T 1 members. So, what I want to do is basically take this and copy to the T 1 member and that is what is written here. How I do it? Naturally to be able to do this, I need the copy constructor of point. I cannot do it with a parameterize constructor or a default constructor of point.

These are not useful because I already have a point and I want to initialize another point. So, this is the point that I was making that when you have data members of user defined type, and then you will require the copy constructor for those data member types to be available, so that you can easily copy them. So, if you do not; for example say point class. If the point class does not have a copy constructor, then this code or this code will not compile because you will not be able to copy the point as it is required here.

(Refer Slide Time: 10:34)

Module 14	Code	Output	Lifetime	Remarks
Partha Pratim Das	<pre>int main() { Rect r1(0, 2, 5, 7); //Rect(int, int, int, int) Rect r2(r1); Point p1(6, 9); //Rect(Point&, Point&) Rect r3(r2); Point p2(3, 5); Point color((3, 5)); Point color((6, 9)); Rect r4(r3); Point d1(3, 5); Point d2(6, 9); Rect r5(r3); Point p3(2, 2); Point color((2, 2)); Rect r6(r5); Point d3(2, 2); Point d4(6, 4); Rect r7(r6); Point d5(6, 0); Point d6(0, 0); Rect r8(r7); Point d7(0, 0); Point d8(0, 0); }</pre>	<pre>Point ctor [0, 2] Point ctor [5, 7] Rect ctor [[0, 2] [5, 7]] Point ctor [6, 9] Point ctor [3, 5] Point color [(3, 5)] Point color [(6, 9)] Rect ctor [[3, 5] [6, 9]] Point dtor [(3, 5)] Point dtor [(6, 9)] Point ctor [2, 2] Point color [(2, 2)] Rect ctor [[2, 2] [6, 4]] Point dtor [(2, 2)] Point dtor [(6, 0)] Point dtor [(0, 0)] Rect dtor [[0, 0] [0, 0]]</pre>	<pre>Point r1 TL_ Point r1 BR_ Rect r1 Point r2 TL_ Point r2 BR_ Rect r2 Point r3 TL_ Point r3 BR_ Rect r3 Point r4 TL_ Point r4 BR_ Rect r4 Point r5 TL_ Point r5 BR_ Rect r5</pre>	Second parameter First parameter Copy to r2.TL_... Copy to r2.BR_...
Objectives & Outline				Fist parameter Second parameter First parameter Copy to r3.TL_...
Lifetime Examples				Fist parameter Second parameter First parameter Copy to r4.TL_...
String				
File				
Matrix & Address				
CardDeck				
Copy Constructor				
Get by value				
Assignment				
Operator				
Class Pointer				
Self Copy				
Suspension				
Summary				

So, this is just an small application between here with different rectangle objects being constructed and subsequently being this destructed and if you look into the table here, we have shown all the different objects that are getting constructed in this process and the life time of which object longs for how long. Again the idea is not to explain it here, but I have worked this out, so that you can take this code carefully, study and convince yourself that you understand exactly how the construction, the copy construction. You will see several copy constructions happening here. You can convince yourself as to why this copy construction happens.

For example, if I construct a point and pass it to a function, it will construct a copy. In our case, it will not be because the point is passed as a reference, but when I set the TL field or the BR field of a rectangle object, then it certainly will lead the copy construction to happen. So, please go through this and convince yourself that you understand the whole process of construction, destruction and copy construction together.

(Refer Slide Time: 11:53).

The slide has a blue header with the title 'Free Copy Constructor'. On the left, there's a sidebar with a logo and a navigation menu. The main content area contains two bullet points about free copy constructors.

Module 14:
Partha Pratim Das
Objectives & Outline
Lecture Examples
String Data Types
Memory & Address
Conditional
Copy Constructors
Call by value
Data members
Free Copy Constructors
Copy Assignment Operator
Late Pointer and Copy Registration
Summary

Free Copy Constructor

- If no copy constructor is provided by the user, the compiler supplies a *free* copy constructor
- Compiler-provided copy constructor, understandably, cannot initialize the object to proper values. It has no code in its body. It performs a *bit-copy*

NPTEL MOOCs Programming in C++ Partha Pratim Das 19

Now, like for the constructor and destructor, we have free versions. The same mechanisms are also available for copy constructor. If no copy constructor is provided by the user, by the programmer, if I write a class which does not have a copy constructor, the compiler will supply a free copy constructor and the compiler provide a copy. Constructor certainly cannot initialize the object because it does not know how to do it. So, what it does is, it simply makes a bit copy which means it takes the complete bit pattern of the object to copy from and makes another bit pattern exactly seem as the object.

Now, you will ask what is wrong in that. That should actually generate a copy. Now, that is where we get into severe problems. We will just illustrate soon that just copying the bits is not copying the object, but if it is same that is if it is copying the bits is copying the object, then we can leave the free copy constructor that the compiler provides and the story is similar. If we provide a copy constructor, then the compiler stops providing one.

(Refer Slide Time: 13:10)

Module 14
Partha Pratim Das
Objectives & Outline
1.1 Introduction
1.2 Examples
1.3 String
1.4 File
1.5 User
1.6 Notes & Actions
1.7 Credits and Acknowledgments
1.8 Copy Constructors
1.9 Call by value
1.10 Data members
1.11 Free Copy Constructor
1.12 Copy Assignment Operator
1.13 Copy-Pasting
1.14 References
1.15 Summary

Program 14.09: Complex: Free Copy Constructor

```
#include <iostream>
using namespace std;

class Complex { double re_, im_; public:
    Complex(double re, double im) : re_(re), im_(im) // Constructor
    { cout << "ctor: " << print(); }
    //Complex(const Complex& c) : re_(c.re_), im_(c.im_) // Copy Constructor
    //{{ cout << "copy ctor: " << print(); }
    //Complex() { cout << "dtor: " << print(); }
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "(" << re_ << "," << im_ << ")" = " << norm() << endl; }
};

void Display(Complex c_param) { cout << "Display: " << c_param.print(); }

int main() {
    Complex c(4.2, 5.3); // Constructor = Complex(double, double)
    Display(c); // Free Copy Constructor called to copy c to c_param
    return 0;
}
```

User-defined CCtor
ctor: 14.2+5j.3i = 6.7624
copy ctor: 14.2+5j.3i = 6.7624
Display: 14.2+5j.3i = 6.7624
dtor: 14.2+5j.3i = 6.7624
dtor: 14.2+5j.3i = 6.7624

Free CCtor
ctor: 14.2+5j.3i = 6.7624
\\ No message from free CCtor
Display: 14.2+5j.3i = 6.7624
dtor: 14.2+5j.3i = 6.7624
dtor: 14.2+5j.3i = 6.7624

- ★ User has provided no copy constructor
- ★ Compiler provides free copy constructor
- ★ Compiler-provided copy constructor performs bit-copy - hence there is no message
- ★ Correct in this case as members are of built-in type

NPTEL MOOCs Programming in C++ Partha Pratim Das 20

So, we are back to the complex class. We are now using the free copy constructor. So, the copy constructor that was written, I have simply made them commented and I am still trying to use the call by value, calling the display function and you will see if you compare with the previous one, this is the one which you have seen earlier and this is the one when we provided the copy constructor earlier. This is the one here, where there is no copy constructor given. The compiler is providing that and the only difference is since no copy constructor is given by us, there is no explicit message saying that the copy is being constructed, but otherwise if you look at the output, it is exactly the same.

So, it does not make a difference which means in other terms in this case copying bits is same as copying the object, but do not generalize. Hold on, do not generalize.

(Refer Slide Time: 14:16).

Module 14
Partha Pratim Das
Objectives & Outcome
Lecture Examples
String
Date
Time
Name & Address
Constructor
Copy
Constructor
Call by value
Data members
Free Copy
Constructor
Copy
Assignment
Operator
Final Points
Self-Copy
Simplification
Summary

```
#include <iostream>
#include <cstring>
#include <cstring>
using namespace std;
class String { public: char *str_; size_t len_;}
String(char *s) : str_( strdup(s)), len_(strlen(str_)) {} // ctor
String(const String &) : str_( strdup(s.str_), len_(s.len_)) {} // ctor
~String() { free(str_); } // dtor
void print() { cout << "(" << str_ << "): " << len_ << ")" << endl; }
void strToUpper(string &s) { // Make the string uppercase
    for (int i = 0; i < s.len_-1; i++) s.str_[i] = toupper(s.str_[i]);
    cout << "strToUpper: (" << s.print() << ")";
}
int main() {
    String s = "Partha";
    s.print();
    strToUpper(s);
    s.print();
    return 0;
}
```

(Partha: 6)
strToUpper: (PARTHA: 6)
(Partha: 6)

• User has provided copy constructor. So Compiler does not provide free copy constructor
• When actual parameter *s* is copied to formal parameter *a*, space is allocated for *a.str_*, and then it is copied from *s.str_*. On exit from *strToUpper*, *a* is destructed and *a.str_* is deallocated. But in *main*, *a* remains intact and access to *a.str_* is valid.
• Deep Copy: While copying the object, the pointed object is copied in a fresh allocation. This is safe

NPTEL MOOCs Programming in C++ Partha Pratim Das 21

Let us consider, let us go back to our string class and we will try to copy. So, if you try to copy, I have written a copy constructor, simple. What the copy constructor will have to do? Simply you will have to make that copy of the string. So, it does another strdup on the string member of the S object, the object to copy from and it copies the length that will suffice in copying the string.

Given this copy constructor, we have also written a simple function which takes a string and converts to upper case. So, I construct a string and print it, you get to say this, then we call strToUpper, then print it again and this is the output that will expect which is pretty much fine and note that in string, we have actually pass this as value. In strToUpper function, we have passed this as a value.

(Refer Slide Time: 15:37)

Module 14
Partha Pratim Das
Objectives & Outline
Journals
Examples
String
File
Recursion
Name & Address
Conditional
Copy
Constructor
Edit by value
Data members
Free Copy
Constructor
Copy
Assignment
Operator
Data Pointer
Copy
Constructor
Summary

```
#include <iostream.h>
using namespace std;
class String { public: char str[]; size_t len; };
String(char *s) : str(s), len(strlen(str)) {} // ctor
~String(const String& s) : str(s.str), len(s.len) {} // dtor
String() { free(str); } // dtor
void print() { cout << str << endl; }
void strToUpper(String& a) { // Make the string uppercase
    for (int i = 0; i < a.len; i++) a.str[i] = toupper(a.str[i]);
    cout << "strToUpper: " << a.print(); }
int main() {
    String s = "Partha";
    s.print();
    strToUpper(s);
    s.print();
    return 0;
}
```

User-defined CCtor
(Partha: 6) strToUpper: (PARTHA: 6)
(Partha: 6)

Free CCtor
(Partha: 6) strToUpper: (PARTHA: 6)
(????????????????????????????: 6)

● User has provided no copy constructor. Compiler provides free copy constructor
● Free copy constructor performs bit-copy - hence no allocation is done for str, when actual parameter is copied to formal parameter s. s.str is merely copied to a.str, and both continue to point to the same memory. On exit from strToUpper, a is destructed and a.str is deallocated. Hence in main access to a.str is corrupted. Program crashes.
● Shallow Copy: With bit-copy, only the pointer is copied - not the pointed object. This may be risky

NPTEL MOOCs Programming in C++ Partha Pratim Das 22

Now, I do the same thing, but only thing is, I comment out the copy constructor. I have commented out the copy constructor. Unfortunately for our convenience, the compiler will not complain. The compiler will say it is fine. You have not provided a copy constructor. The free copy constructor would be there which will copy the bit pattern and therefore, the same program will compile correctly and this is the output with the user define copy constructor, but when I run this program, I get this kind of an error. This is fine; but just look at these point.

I get some garbage output and then, the program crashes. Now, you need to understand a little bit in terms of what is happening here is, if you look into the original object, this is how the original object looked. So, this is what goes to Partha, this is the str and it goes to Partha. This is len which is x.

Now, since I do a bit copy, so I get another object which has exactly the same set of bits which mean it has exactly the same pointer address as in the original one. So, in my copied object also, I have the same values 6 for length and I have the same object we used here. When I called the upper, I made this. So, this was s. I am calling the upper here. This is s and the formal parameter is a. So, this is a, and why did it copy like this

because I have not provided a copy constructor. If I had done one as I did before, then it would have looked like this s which has Partha 6.

Then it is ‘a’ which has Partha because I had explicitly deducted strndup of this string into the new object string, but because I did not provide a copy constructor, the bit patterns have got copied. So, naturally pointers are same. They are pointing to the same thing. So, this is what happens when I am here. When I have entered the function strToUpper, naturally it has a valid string. So, it will take things to upper case, it takes things to upper case, prints it, fine.

What happens at this point is, these are points where the scope of strToUpper ends. We know that, this is an automatic object and its scope ends at this point, so that distractor of string gets called. What does a distractor do? It freezes string. So, at this point what happens is the distractor of A is called. So, this will free up this string. So, this memory goes and I am back here at this point. I am back here. Now, what happens if this memory has gone? Then for s, the str now points to nothing. Whatever it used to point to that address is with us, but at that place, there is no string available because I have already freed that to the system disaster to say the least.

That is the reason when I am trying to print them, it prints some garbage characters. It is not necessary that it will print question mark. I have just put question mark as a place holder. I can print anything because it just does not know where it is printing from and it results into this kind of an error of the program crash.

So, this example clearly shows you that bit copy is not same as object copy and that gives us two different notion of copy. When an object has pointers that are referring to other objects that are dynamically created, then while copying it, if we just copy the pointer, but do not copy the object, then we say that we are doing a shallow copy, but in contrast while copying the object, if we do not copy the pointer, but you copy the pointed object, then we will say that we have done a deep copy.

So, while we wrote the copy constructor for the string, we did a deep copy which was fine because when stroupper is called by deep copy, a different str data was created and

that created str data got destructed when strToUpper function finished, but when we did not do that, when we allowed the compiler to provide the free copy constructor, then it resulted in a shallow copy. So, when the local parameter or the parameter of strToUpper was destructed, the original actual parameter also lost this value. This is the kind of problem that you can get into if you do not distinguish between shallow and deep copy.

(Refer Slide Time: 21:44)

```

Module 14
Partha Pratim Das

#include <iostream>
#include <cmath>
using namespace std;

class Complex { double re, im; public:
    Complex(double re_, double im_) : re(re_), im_(im) { cout << "ctor: " << print(); }
    Complex(const Complex& c) : re(c.re), im(c.im) { cout << "cctor: " << print(); }
    ~Complex() { cout << "dtor: " << print(); }
    Complex operator=(const Complex& c) // Copy Assignment Operator
    { re_ = c.re; im_ = c.im; cout << "copy: " << print(); return *this; }
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "(" << re_ << "," << im_ << ")"; cout << endl; }
};

int main() {
    Complex c1(4.2, 6.3), c2(7.9, 8.6); // Constructor - Complex(double, double)
    Complex c3(c1); // Constructor - Complex(const Complex& c)

    c1.print(); c2.print(); c3.print(); // Copy Assignment Operator
    c1 = c3; c1.print(); c2.print(); c3.print(); // Copy Assignment Chain
    return 0;
}

ctor: (4.2+6.3i) = 6.7624 // c1 = ctor
ctor: (7.9+8.6i) = 11.6043 // c2 = ctor
cctor: (7.9+8.6i) = 11.6043 // c3 = cctor
14.2+56.3i = 6.7624 // c1
17.9+8.6i = 11.6043 // c2
17.9+8.6i = 11.6043 // c3
copy: (4.2+6.3i) = 6.7624 // c2 = c1
copy: (4.2+6.3i) = 6.7624 // c2 = dtor
14.2+56.3i = 6.7624 // c2

```

★ Copy assignment operator should return the object to make chain assignments possible

Partha Pratim Das 23

The next that we want to look at is known as copy assignment. In copy assignment let me first show you the example which, I am sorry, that is some objects have been created. These objects are created directly. C3 is created by copy construction. We have printed so far is what you already know, but then now what you are trying to do is to assigns C1 into C2, that is, this is what for built in types. We also make copies in object terms. This needs to be distinguished from the copy construction because when I write this C2 is being copied into C3, but with the fact that C3 does not exist, C3 has to get created and while it gets created, it must be a copy of C2 whereas if you consider this assignment, C1 being assigned to C2 then, C2 already exist, C1 also exist.

Naturally I want to take C1 and make C2 a copy of C1. So, both cases have copy, but with a fundamental difference that, in this case, there was no earlier object and the object is created by copy and this case the object existed and we are nearly changing the data

members of the object from taking values from another object of the same time. So, this we said is the copy construction. This we say is the copy assignment or in some cases we said this is a simple assignment.

So, naturally if copy assignment has to be done, then we need this kind of an operator to be present. This is known as the copy assignment operator. If you recall the operator over loading discussions earlier we have discussed that every operator has a corresponding operator function and the assignment operator has this function because what I am assigning, I am assigning C2 being assigned C1. So, this is equivalent to saying that C2 dot operator assignment C1. So, the function corresponding to the assignment operator is this function. It takes C1 which necessarily is an object of the same time and what does it certainly test an object of the same type.

Why certainly is questionable? It could have returned nothing also as long as it does the copy, but we will discuss why it should also return an object of the same type and in the process, it will copy the respective data members as we want.

So, whenever we write this, this operator will get invoked and according to this operator, the assignment will happen which will mean that in this logic I can decide what I want to copy. So, it is notionally a copy always means that it is kind of a clone, right. We loosely mean that it is a clone that it is identical to whatever I am copying from, but in actual terms of C++, a copy is not necessarily a clone. A copy is whatever I want to copy. I may not want to copy everything, I may want to copy some part, or I may want to do anything because I am actually being able to write an operator function for the copy.

Now, to answer the question it is easy to understand that why the parameter to a copy assignment operator has to be of the same type as of with class. So, it has to be complex. It is easy to understand as to why this should be reference because if this were not a reference, then you will unnecessarily require a copy construction to happen and then the assignment will happen.

So, that is not what you would want to do. It is understandable that if it is a reference, why should it be constant? It should be because while you are copying, you do not want

the right hand side to change. That is a common semantics. So, this part is clear. What is not obvious is, why this return an object of the same type? For that, just consider the next line in the example. What is C1? What is C1 assigned C2 assigned C3.

Please recall your C. This is a case where the associativity is right. It means that it happens from right to left. So, this is equivalent to C1 assigned C2 assigned C3. This is equivalent value which means the assignment of C3 to C2 must be an expression always as a value, right. It must be an expression and it must be such an expression, so that I can assign it to C1 which means whatever is the written type here that must be able to go as a parameter type of the same operator. If i do not ensure of that, I will not be able to write this kind of a chain assignment. I will be able to write one level of assignment C1 assign C2.

If operator assignment simply returns a wait or something else, then I will be able to still write this because by this I have made changes to the object from C1 from the object C2. So, it will give me the same effect, but I will not be able to write this chain assignment. That is a reason the copy assignment operator always has the same type. That is a reference to the class as input or parameter and reference to that class as a return type for the output, so that you can make this kind of a change assignment possible.

Now, having said that if you now go through this, we can quickly go through; these are three constructions. These are the normal constructions and these are copy construction, identical object coming from here. These three are the three print statements are from here. This is making a copy.

In the copy assignment operator, we have specifically written what copy clone, so that you can know that this is what is happening from your copy assignment operator. So, this makes a copy of C1 to C2 and then, you can see the print shows what is C2 and then these two where first C2 gets assigned to 3, then the result gets assigned to C1 and then, they prints and naturally the reverse order of their destruction. So, this will clearly show you that process of copy assignment for different objects in a class.

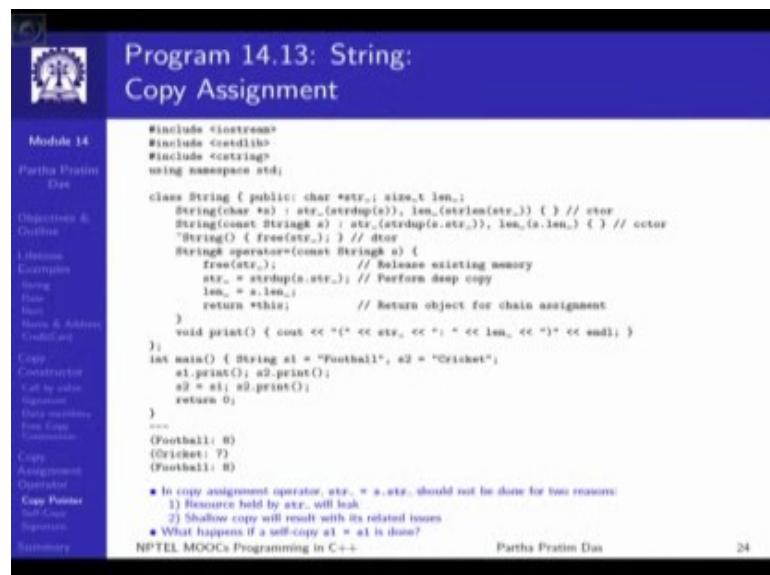
Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 28
Copy Constructor and Copy Assignment Operator (Contd.)

Welcome back to module 14 of Programming in C++. We have been discussing about copy, we have discussed about copy construction at depth and we have introduced what is copy assignment operator.

To quickly recap, we do a copy construction when you want to make a clone of an object which does not exist and we do copy assignment, when we have an existing object and want to copy another object of the same type into this existing object. We have seen that a copy assignment can be defined as a function; operator function and it takes the parameter as a constant reference of the class and it returns a non-constant reference of the same class, it may return a constant reference of the class as well.

(Refer Slide Time: 01:19)



Module 14
Partha Pratim Das
Objectives &
Outline
Lecture
Examples
String
Date
List
Hash & Address
Container
Copy
Constructor
Call by value
Signature
Data members
Free Copy
Assignment
Copy
Assignment
Operator
Copy Pointer
Self Copy
Assignment
Summary

```
#include <iostream>
#include <cstdlib>
#include <cstring>
using namespace std;

class String { public: char *str_; size_t len_;}
String(char *s) : str_(strdup(s)), len_(strlen(str_)) {} // ctor
String(const String& s) : str_(strdup(s.str_)), len_(s.len_) {} // copy ctor
~String() { free(str_); } // dtor
String& operator=(const String& s) {
    free(str_);
    str_ = strdup(s.str_);
    len_ = s.len_;
    return *this; // Return object for chain assignment
}
void print() { cout << "(" << str_ << ", " << len_ << ")" << endl; }
int main() { String s1 = "Football"; s2 = "Cricket";
    s1.print(); s2.print();
    s2 = s1; s2.print();
    return 0;
}

```

• In copy assignment operator, `str_ = s.str_`, should not be done for two reasons:
1) Resource held by `s.str_` will leak.
2) Shallow copy will result with its related issues
• What happens if a self-copy `s1 = s1` is done?

NPTEL MOOCs Programming in C++ Partha Pratim Das 24

Now, we will look into some of the more tricky areas of copy assignment. So, you recall that in terms of copy construction, you have already explained the notion of shallow

copy and deep copy. Now we will see that the consequences of shallow copy and deep copy also percolates into copy assignment. So, particularly focus this is a string example and focus on the copy assignment operator.

Now, what you are trying to do? This is your; let me just draw the two objects. This is `s1`, this is `s2`. So, this is football. So, whatever it will have, it will have a length 8. This is cricket; this will have length 7. Now, I am trying to copy. So, I am trying to do `s2 = s1`. So, if I copy this; naturally while I do copy I know that this string will have to be copied here and we know there are two options to that. One is to copy the pointer; other is to actually copy the object. So, we would like to do deep copy, in this case we are doing a deep copy. So, you have making `strdup` of a parameter `s`, which is basically `strdup` of `s1`.

Now, when you assign that to `str` then what will happen? Simply, so another football has got created, `strdup` we have done duplicate. So, another football has got created. Now, if I put this pointer into `str`, naturally I will lose this pointer and there will be no way to retrieve this string any further. So, before I can do this, I have to free up, otherwise the resource will leak, memory will leak. So, this is a critical point. I will first have to free this up and then I am doing a string copy. So, as I free this up this is gone, I do a `strdup`. Now, I have a new football pointed to here, this will get copied. So, this will become 8 as in here and the objective is returned as it was done in the last case.

Just note that this is basically returns a starts this is a current object. So, it returns that because it has to return the object to which the assignment is happening. So, this object can now be used for chain assignment, as I have explained. This can be used in the chain assignment as I have explained already. So, this is how a copy can be done for a string with deep copy that is similar strategy can be used whenever we have pointer members in the object.

(Refer Slide Time: 05:01)

Module 14
Partha Pratim Das

Objectives & Outcome
1. Basics
2. Examples
3. String
4. File
5. Recursion
6. Name & Address
7. Condition
8. Copy
9. Constructors
10. If by value
11. If by reference
12. Data members
13. Function
14. Copy Assignment Operator
15. Case Pointers
16. Self-Copy
17. Inheritance
18. Summary

```
#include <iostream>
#include <cstdlib>
#include <cstring>
using namespace std;

class String { public: char *str_; size_t len_;
```

The code continues with the constructor, destructor, copy assignment operator, and print function, followed by a main() function that prints "Football" and "Cricket". A note at the bottom explains the bug: "For self-copy s1_ and a.s1_ are the same pointers. Hence, free(s1_) first releases the memory, and then strcpy(a.s1_) tries to copy from released memory. This may crash or produce garbage values. Self-copy must be detected and protected."

NPTEL MOOCs Programming in C++ Partha Pratim Das 25

Now, let's look into a very small, but dangerous issue with the code that we have already seen. This is the code exactly that you have seen, the only difference being earlier we were coping s2 to s1. Now, I have copied s1 to s1. Now, you can very legitimately ask me, as to why should somebody write this kind of a code, there are two answers to that; one is what if somebody writes. We have to know, what is going to happen. The other issue is that not always the code will look like this, for example, it could be that; I have a string, I have a reference to s1 that is done somewhere; I do not know where this is done.

This may have been done in some other function, in some other class whatever it is come and now I am doing s1 assigned r. Syntactically, looking at the code it does not look like a self copy, but it actually the self copy. So, self copy is a potential situation that we must look into. Now certainly there are issues that the reason we are trying to look into this.

So, look into this is the self copy. So, this is what I have, this is my s1, this is my string, my s1 is football. So, I have football and a weight here. Now, I am doing s1 assign this one. So, what will happen? This will execute first, this is my s1. So, this will free this up. Now, this will try to do this, that is, it will try to take this object s dot str, make a copy make a copy into something and then assign it here. Now, this object is already gone this

is been freed up. So, what you make copy of here is not known, is not question mark it is just not known, it is something invalid and then so on.

So, quiet expectedly what you get after the copy, when you print it after the copy you get a garbage, I got a garbage, while I was running it, but it is quite possible that instead of a garbage, it could be a crash because it just depends on what memory is getting violated. So, self copy with pointer type of data is something which could prove to be quite difficult to deal with. So, we will have to do something about that.

(Refer Slide Time: 08:10)

```

Module 14
Partha Pratim Das
Directions & Outline
Lifetime Examples
Using
One
More
Point & Address
CreditCard
Copy Constructors
Call by value
Signature
Data members
Data Copy Constructors
Copy Assignment Operator
Copy Pointer
Self Copy
Implementation Summary

```

**Program 14.14: String:
Self Copy – Safe**

```

#include <iostream>
#include <stdlib.h>
#include <string.h>
using namespace std;

class String {
public:
    char *str;
    size_t len;
};

String::String(const char *s) : str((char*)malloc(strlen(s))), len_(strlen(s)) {} // ctor
String::String(size_t s) : str((char*)malloc(s)), len_(s), len_s(s) {} // ctor
String::String() : str(0), len_(0), len_s(0) {} // dtor
String::operator=(const String &s) {
    if (this != &s) {
        free(str);
        str = strdup(s.str);
        len_ = s.len;
    }
    return *this;
}
void print() { cout << "(" << str << ", " << len_ << ")" << endl; }

int main() {
    String s1 = "Football";
    s2 = "Cricket";
    s1.print();
    s2.print();
    s1 = s1;
    s1.print();
    return 0;
}

/*
(Football: 8)
(Cricket: 7)
(Football: 8)

• Check for self-copy (this != &s)
• In case of self-copy, do nothing
NPTEL MOOCs Programming in C++

```

So, the way we handle this and that is very typically is all that you want to say that if I am doing a self copy, if I am doing this, then all the time need to tell my copy assignment operator is that do not copy. If you are doing a self copy then all that I would like to tell is do not copy because it is bypass, it is a same object. So, the rest of the code remains same, but all that I add is check, if it is a same object. How do I check if it is a same object? Just understand s1 is being assigned s1. So, it is s1 dot operator assignment s1, this become s and this is the object on which the invocation has happen, so this is * this.

So, you want to see whether star this and s, are same. We cannot compare objects like that because this could be any object, I do not have a comparison operator for that. These

are not like integer that I can write equal to equal to but what all that I know is, if it is a same object then it resides in the same memory. So, if these 2 have to be same then this has to be same as &s. Their addresses have to be same, if the addresses are same that is the same object; if the addresses are different they are not the same object.

So, all that you simply do is check, if the addresses are different. If the addresses are different, you go through the copy if it is not then you simple by pass. So, these are small point about self copy in copy assignment operator that you should always keep in mind and this is a typical way to write a copy assignment operator, particularly in the cases where you have pointer type of data members.

(Refer Slide Time: 10:14)

The slide is titled "Signature and Body of Copy Assignment Operator". It features a sidebar with a logo and a list of topics: Module 14, Partha Pratim Das, Objectives & Outline, Lifetime Examples, Using Data, Data & Address, Overload, Copy Constructor, Call by value, Signature, Data members, Free Copy Construction, Copy Assignment Operator, Copy Pointer, Self Copy Signature, and Summary. The main content area contains three bullet points:

- For class MyClass, typical copy assignment operator will be:

```
MyClass& operator=(const MyClass& a) {  
    if (this != &a) {  
        // Release resources held by *this  
        // Copy members of a to members of *this  
    }  
    return *this;  
}
```
- Signature of a Copy Assignment Operator can be one of:

```
MyClass operator=(const MyClass& rhs); // Common  
const MyClass operator=(const MyClass& rhs); // Occasional  
MyClass operator=(MyClass& rhs); // Change in Source
```
- The following Copy Assignment Operators are occasionally used:

```
MyClass operator=(MyClass rhs);  
const MyClass operator=(const MyClass rhs);  
const MyClass operator=(MyClass& rhs);  
const MyClass operator=(MyClass rhs);  
MyClass operator=(const MyClass rhs);  
MyClass operator=(MyClass& rhs);  
MyClass operator=(MyClass rhs);
```

The signature of the copy assignment operator, we have already seen. This is a typical signature and this is a basic structure that we have shown you first check for self copy then you release any resource that is currently held by the object being assigned to and then copy the rest of the members to the current object. It can be one of this, that is, it is also possible that you do not use const you just do a copy without a constant on the parameter.

So, which means the during copy actually the object you are coping from can get changed and we will see that this has a very serious use in terms of the design, particularly in some smart designs known as smart pointers, where this particular feature will be used extensively, but will talk about that when the time comes and there are several other signatures; which I just listed them. Do not spend a lot of effort to understand or to memorize what these are allowed and these are used occasionally, but they are very, very occasionally in situation. So, it is just that such copy assignment operators are possible, but you will primarily use this and in some cases you will use this.

(Refer Slide Time: 11:53)

The slide is titled "Module Summary" and features a video thumbnail of a man in a blue shirt. The left sidebar contains a navigation menu with items like "Module 14", "Partha Pratim Das", "Objectives & Outcome", "Lecture Examples", "String", "Date", "Date", "Name & Address", "CreditCard", "Copy Constructor", "Call by value", "References", "Data members", "Free Copy Construction", "Copy Assignment Operator", "Copy Pointer", "Call by Reference", and "Summary". The main content area lists three bullet points under the heading "Module Summary":

- **Copy Constructors**
 - A new object is created
 - The new object is initialized with the value of data members of another object
- **Copy Assignment Operator**
 - An object is already existing (and initialized)
 - The members of the existing object are replaced by values of data members of another object
- **Deep and Shallow Copy for Pointer Members**
 - Deep copy allocates new space for the contents and copies the pointed data
 - Shallow copy merely copies the pointer value – hence, the new copy and the original pointer continue to point to the same data

At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" with page number "28".

So, to sum up here we have looked into copy constructors, where new object is created, and this new object is initialized with the value of the data members of another object and the major requirement of copy construction happens for call by value and for initializing use a defined type data members. Copy constructors are to be provided by the user, but if the user does not provide a copy constructor then the compiler will provide a free copy constructor which just as a bit copy. We have discussed about copy assignment operator which is doing a copy when the object already existing.

So, if it is already existing initialized then it has to be replaced by the members of the object being copied from, and in copy assignment operator self copy could be a significant issue and needs to be taken care of. Again please keep in mind that this is not explicitly written in the slide, but please keep in mind that if the user does not provide a copy assignment operator, but uses it in the program then the compiler will provide a free copy assignment operator, which again like; the free copy constructor will again just do a bit wise copy without considering what specific requirements the copy may have.

So, it is always advised that like the constructor, you should also provide the copy constructor and the copy assignment operator whenever you are designing a class where copies are possible or where objects are likely to be passed to functions in call by value.

In specific terms, we have also seen here the notions of the deep and shallow copy with pointers. Please remember, shallow copy will just copy the pointer. So that after a shallow copy more than one pointer points to the same object and deep copy does not copy the pointer it copies the pointed object. Therefore, after deep copy the 2 pointers point to two different copies after possibly this originally same object, but they become different objects.

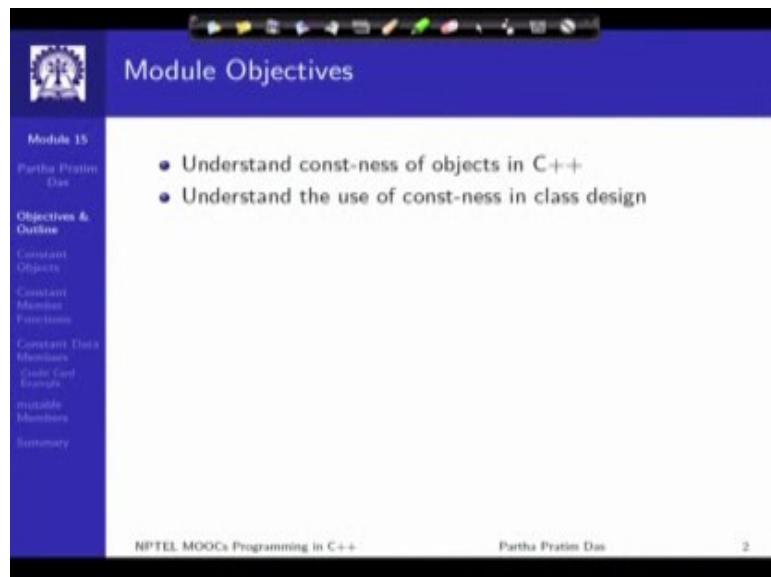
So, deep copy and shallow copy will have to be used naturally judiciously. Certainly, if it is not required, we will not try to do deep copy because it will involve the copying of the pointed data which may be costly because that will again need copy construction by recursive logic, but in terms of safety, using deep copy is often more safe compared to using shallow copy.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 29
Const-ness

Welcome to module 15 of Programming in C++. In this module, we will discuss about const-ness.

(Refer Slide Time: 00:36)



Module Objectives

Module 15
Partha Pratim Das

Objectives & Outline

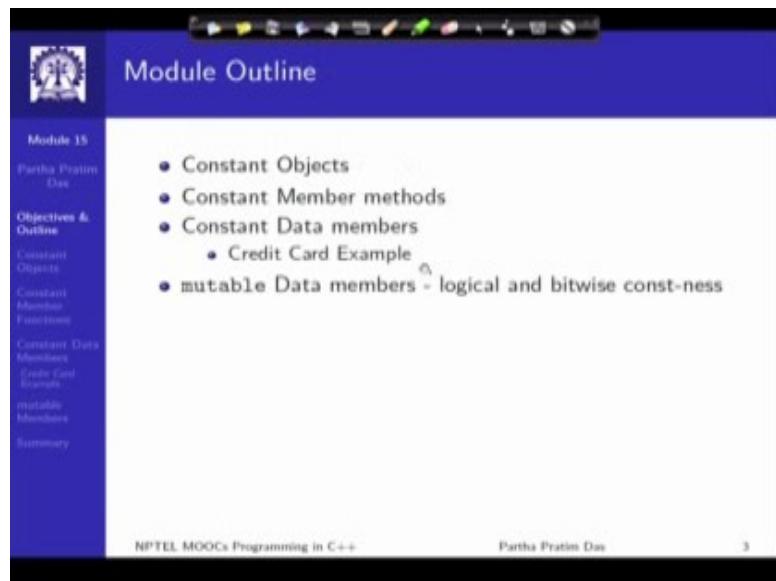
- Constant Objects
- Constant Member Functions
- Constant Data Members
- Const Cast Examples
- mutable Members
- Summary

- Understand const-ness of objects in C++
- Understand the use of const-ness in class design

NPTEL MOOCs Programming in C++ Partha Pratim Das 2

To understand the const-ness of C++, objects will be the main objectives for this module and we will also see how const-ness can be used in class design.

(Refer Slide Time: 00:51)



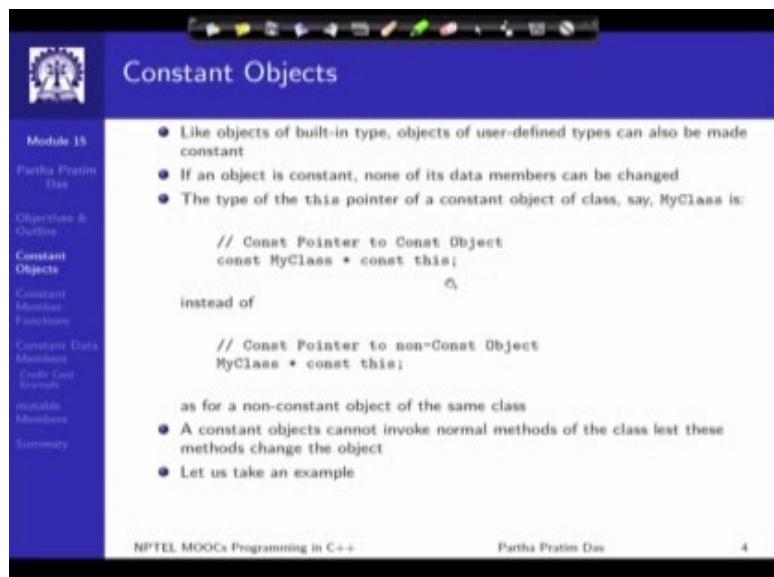
The slide shows a module outline titled "Module Outline". The left sidebar lists topics: "Partha Pratim Das", "Objectives & Outline", "Constant Objects", "Constant Member functions", "Constant Data members", "Credit Card Example", "mutable Data members - logical and bitwise const-ness", and "Summary". The main content area contains a bulleted list:

- Constant Objects
- Constant Member methods
- Constant Data members
 - Credit Card Example
- mutable Data members - logical and bitwise const-ness

At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" with a page number "3".

The outline is as given, it will follow on the left hand side. We will talk about constant objects; member functions; data members take an elaborate example and also discuss mutable data members.

(Refer Slide Time: 01:08)



The slide is titled "Constant Objects". The left sidebar lists topics: "Partha Pratim Das", "Objectives & Outline", "Constant Objects", "Constant Member functions", "Constant Data members", "Credit Card Example", "mutable Data members", and "Summary". The main content area contains a bulleted list:

- Like objects of built-in type, objects of user-defined types can also be made constant
- If an object is constant, none of its data members can be changed
- The type of the `this` pointer of a constant object of class, say, `MyClass` is:

```
// Const Pointer to Const Object
const MyClass * const this;
```

instead of

```
// Const Pointer to non-Const Object
MyClass * const this;
```

as for a non-constant object of the same class

- A constant objects cannot invoke normal methods of the class lest these methods change the object
- Let us take an example

At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" with a page number "4".

First, let us introduce constant objects. We have seen const-ness earlier when we talked

about the procedural extension of C++, the better C++ part. We had seen that if we make an object constant, or a data constant, a variable constant then that cannot be changed; that has been set a value at the initialization time and after that, that value cannot be changed. So, this we had seen for built-in type data - int, double, the pointer and so on.

Now the same thing can be done with C++ objects. So, we note that like objects in a built-in type, the user define type objects can also be made constant. Now, naturally if an object is made constant then none of its data members can be changed; An object has one or more data members. Once you make it constant none of them can be changed.

The primary consequence of making an object constant is that type of it is this pointer. We remember that this pointer refers to the address where this object reside, this pointer now becomes a pointer to a constant object. So, earlier when we introduced this pointer, we saw this is the type of this pointer, it is a constant pointer to the object of the class, but now it becomes a constant pointer to a constant object of the class. So, introduction of this const in the type of this pointer is a consequence of have declaring an object as constant.

Naturally, a constant object cannot invoke the normal methods of the class. This is understandable because we are saying that a constant object should not be allowed to change the data members of the object. Now, instead of directly changing, if a method is invoked then that method can in turn change the data member. Therefore, to avoid that constant objects are not allowed to invoke the data member, we invoke the member functions. We will see, inspite of this how constant objects can actually function.

(Refer Slide Time: 03:43)

The screenshot shows a presentation slide titled "Program 15.01: Example: Non-Constant Objects". The slide content is as follows:

```
#include <iostream>
using namespace std;

class MyClass {
    int myPriMember_;
public:
    int myPubMember_;
    MyClass(int nPri, int nPub) : myPriMember_(nPri), myPubMember_(nPub) {}
    int getMember() { return myPriMember_; }
    void setMember(int i) { myPriMember_ = i; }
    void print() { cout << myPriMember_ << ", " << myPubMember_ << endl; }
};

int main() {
    MyClass myObj(0, 1); // Non-constant object
    cout << myObj.getMember() << endl;
    myObj.setMember(2);
    myObj.myPubMember_ = 3;
    myObj.print();
}

non
0
2, 3
```

At the bottom of the slide, there is a note:

- It is okay to invoke methods for non-constant object `myObj`.
- It is okay to make changes in non-constant object `myObj` by method (`setMember()`).
- It is okay to make changes in non-constant object `myObj` directly (`myPubMember_`).

The slide also includes the footer information: "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

So, let us first take an example to understand this. So, here is an example of a non constant object. So, this non constant object has two members, one that is a private. So, this non constant object has a private member and a public member. So, we have introduced two member functions; get member and set member to read and write the private member and the public member naturally cannot be directly read or written. So, here is an application which is trying to read the private member, write the private member, read the public member and over all print the object. So, all of these will work fine you already understand this very well.

(Refer Slide Time: 04:43)

Module 15
Partha Pratim Das
Objectives & Outline
Constant Objects
Constant Object Functions
Constant Data Members
Friends and Friends
mutable Members
Summary

```
#include <iostream>
using namespace std;

class MyClass {
    int myPriMember_;
public:
    int myPubMember_1;
    MyClass(int aPri, int aPub) : myPriMember_(aPri), myPubMember_(aPub) {}
    int getMember() { return myPriMember_; }
    void setMember(int i) { myPriMember_ = i; }
    void print() { cout << myPriMember_ << ", " << myPubMember_ << endl; }
};

int main() {
    const MyClass myConstObj(5, 6); // Constant object

    cout << myConstObj.getMember() << endl; // Error 1
    myConstObj.setMember(7); // Error 2
    myConstObj.myPubMember_ = 8; // Error 3
    myConstObj.print(); // Error 4

    return 0;
}
```

• It is not allowed to invoke methods or make changes in constant object myConstObj
• Error (1, 2 & 4) on method invocation typically is:
 cannot convert 'this' pointer from 'const MyClass' to 'MyClass &'
• Error (3) on member update typically is:
 'myConstObj': you cannot assign to a variable that is const
• With const, this pointer is const MyClass * const while the methods expects MyClass
• Consequently, we cannot print the data member of the class (even without changing it)
• Fortunately, constant objects can invoke (select) methods if they are constant member functions

NPTEL MOOCs Programming in C++
Partha Pratim Das

Now, we take a constant object. So, the same example, the difference that we are making is now; you have made the object constant. So, before the declaration of the object we have written const. So, by this the object my const obj is become a constant object, rest of the class is the same. It is just the use of the object which we have made constant, now with that you will notice that all these four use of the object gives compilation error. So, in the first case it is trying to invoke a get member to read the private member and print it. Now, this gives an error as you can see here, it says that cannot convert this pointer from const myClass to myClass &.

We will understand little bit more of what it really means, but to take the bottom line, the compiler is saying that since the object is constant, you cannot use it to invoke a member function. The same error is obtained in this case of set member, in case of print error 1, 2 and 4, all of them give the same error and if we try to directly write to the public member as we are trying to do here, there is a public member. So, if we try to directly write to this public member then we get a different error, which says that you cannot assign to a data member when the object is constant. So, this ensures that if the object is been declared as constant then we cannot access or change the data member's member functions of that particular class through that constant object.

Now, naturally on one side of this achieves the objective because it has been able to protect the data members, being constant it has to protect the data members that they cannot be changed, but now on other side it basically defeats the purpose because, for example, if I just wanted to read like in here, I just wanted to read the private member, I am not being able to even to do that or I just wanted to print the object. I cannot even do that because I cannot invoke any member function. So, we need another support in C++ to be able to do this, that is, for constant member functions, for constant objects we need to introduce constant member functions. We will see how this will be done.

(Refer Slide Time: 07:39)

The slide has a blue header bar with the title 'Constant Member Function'. On the left, there's a sidebar with a navigation menu:

- Module 15
- Partha Pratim Das
- Objectives & Outcomes
- Constant Objects
- Constant Member Functions** (this is the current slide)
- Constant Data Members
- Code Examples
- Keywords
- mutable
- Summary

The main content area contains the following text and code snippets:

- To declare a constant member function, we use the keyword `const` between the function header and the body. Like:


```
void print() const { cout << myMember_ << endl; }
```
- A constant member function expects a `this` pointer as:


```
const MyClass * const this;
```

 and hence can be invoked by constant objects
- In a constant member function no data member can be changed. Hence,


```
void setMember(int i) const
{ myMember_ = i; } // data member cannot be changed
```

 gives an error
- Interesting, non-constant objects can invoke constant member functions (by casting – we discuss later) and, of course, non-constant member functions
- Constant objects, however, can only invoke constant member functions
- All member functions that do not need to change an object must be declared as constant member functions

At the bottom, the footer says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

So, constant member functions are a new feature to support const-ness in C++. It is just like another member function, the only difference being that in the constant member function after the function prototype the header; this is the header part and this is the body part of the function. In between these two, you write the keyword `const` and when you write this keyword `const`, the member function is called a constant member function.

Now, what it does, as we all know that when an objects wants to invoke the member function of the class then the address of that object get passed as a `this` pointer in that member function and we known the type of this pointer. Now, when you declare the constant member function then this pointer that is passed gets changed in its signature. It

is now, a pointer to a constant object because you are saying that the member function is a constant member function, so what compiler wants to emphasize is; only constant objects can invoke this member function.

So, we have already seen that the constant objects have this pointer feature of this type, which are constant points is to constant objects and constant member function need this pointer of the same type. So, now, it is possible that a constant object would be able to invoke this constant member function.

The consequence of that is, what is given here, if a member function is constant then no data member can be changed in that function. So, if we have the set member function and if we declare that to be a constant member function. So, that a constant object can invoke it then we will get a compilation error because in this function we are trying to change a data member by making an assignment to it. So, to sum up a constant member function can be used by constant objects only to read data members, only to access data members, but they cannot be used to write or change the data members and in any case non constant member functions cannot be invoked by the constant object.

So, with the combination of these two, we achieve the objective that a constant object can only maintain the data members that it already has maintains the same values, but it cannot allow to change the values of the data members.

Now, the next point here is also interesting that a non constant object can invoke a constant member function. This is simple because a constant member function guarantees that it will not make change to an object, if the object itself is non constant then we do not care whether it is changed or it is not changed. So, you are saying that it is a non constant object and a constant member function invoked on it, says that, I will not make any changes there is no contradiction. So to sum up, a non constant object can invoke constant member function as well as non constant member function, but a constant object can invoke only constant member function.

So, all member functions that do not need to change an object, those must be declared as constant member function to make sure that any constant object can also use them, can

also invoke those member functions.

(Refer Slide Time: 11:58)

Program 15.03: Example Constant Member Functions

```
#include <iostream>
using namespace std;

class MyClass {
    int myPrimeMember_;
public:
    int myPubMember_;

    MyClass(int apri, int apub) : myPrimeMember_(apri), myPubMember_(apub) {}

    int getMember() const { return myPrimeMember_; }

    void setMember(int t) { myPrimeMember_ = t; }

    void print() const { cout << myPrimeMember_ << ", " << myPubMember_ << endl; }
};

int main() {
    MyClass myObj(0, 1); // Non-constant object
    const MyClass myConstObj(5, 6); // Constant object

    cout << myObj.getMember() << endl;
    myObj.setMember(2);
    myObj.myPubMember_ = 3;
    myObj.print();

    cout << myConstObj.getMember() << endl;
    //myConstObj.setMember(7);
    //myConstObj.myPubMember_ = 8;
    myConstObj.print();
    return 0;
}
```

Output

0
2, 3
5
5, 6

NPTEL MOOCs Programming in C++ Partha Pratim Das

Let us, look at an example again. We will get back to the same example, now what we are doing, we will look at the member functions and decide whether they should be constant or they should not be constant; get member is only reading a value so you make it constant, print is only reading and printing values of data members will make it constant, but this one, the set member we cannot make a constant member function because set member intends to change the data member. If we have two objects now, one non constant and one constant, then as you can see that the non constant object can make all the changes as we have seen before.

Constant object now can invoke the get member function because it is a constant member function and read the value of myprime member data. It can print because print is a constant member function, but it cannot do any of these, that is why I have commented them. it neither can invokes set member to change the value of the private member; because set member is a non constant member function cannot be invoked by the constant object and of course, it cannot directly make assignments to the public member because it is a constant object. So, this, we achieve the objective that we can control exactly which a member functions can be invoked by the constant object and

protect all the values of the data members as they have been set at the time of construction.

(Refer Slide Time: 13:52)

The slide has a blue header with the title 'Constant Data members'. On the left, there's a sidebar with a navigation menu:

- Module 15
- Partha Pratim Das
- Objectives & Outline
- Constant Objects
- Constant Member Functions
- Constant Data Members
- Credit Card Example
- mutable Members
- Summary

The main content area contains a bulleted list of points:

- Often we need part of an object, that is, one or more data members to be constant (non-changeable after construction) while the rest of the data members should be changeable. For example:
 - For an Employee: employee ID and DoB should be non-changeable while designation, address, salary etc. should be changeable
 - For a Student: roll number and DoB should be non-changeable while year of study, address, gpa etc. should be changeable
 - For a Credit Card: card number and name of holder should be non-changeable while date of issue, date of expiry, address, cvv number gpa etc. should be changeable
- Do this by making the non-changeable data members as constant
- To make a data member constant, we need to put the const keyword before the declaration of the member in the class
- A constant data member cannot be changed even in a non-constant object**
- A constant data member must be initialized on the initialization list**

At the bottom, there's a footer with 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

Now, let us introduce a third type of const-ness that C++ supports. We have just talked about constant objects, where if I make an object constant then the whole of that object is constant. I just cannot make any change to that object, none of the data members can be changed, but often what we will need is, we will need to change some data members while some other data members should remain constant and I have mentioned several examples, for example, here making an employee record and for that employee record you have an employee id, you have a date of birth.

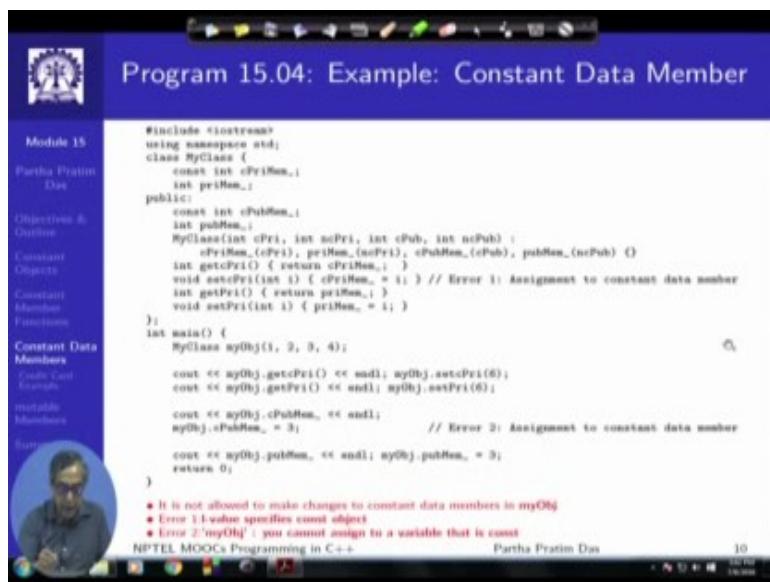
Certainly, once a record has been created we do not expect the employee's id to be changed and of course, date of birth cannot change, but the same record same object will have designation, address, salary, all these they should be changeable because that is what the programming is all about.

So, several other examples if I am dealing with a student then again roll number, date of birth should be non-changeable rest should be changeable, if we were talking about the credit card example we discussed earlier then the card number and the holder of the card

should be non changeable, whereas the cards do have a certain period of issue. So, once those are over, we would like to issue, re-issue the card again. So, the date of issue the date of expiry, the cvv all these will change even the holder can apply and change for an address. So, this should be changeable.

So, constant data members are a feature to support this kind of selective const-ness is a part of the object. So, we do this by making the data members constant by using a const keyword in front of its declaration. So, a constant data member cannot be changed even when the object is not a constant. Of course, if an object is constant then nothing can be changed, but even if an object is non constant then the constant data member will not able to change its value therefore, it must be set at the time of initialization of the object.

(Refer Slide Time: 16:10)



```
#include <iostream>
using namespace std;
class MyClass {
    const int cPriMem_1;
    int priMem_1;
public:
    const int cPriMem_1;
    int pubMem_1;
    MyClass(int cPri, int npri, int cPub, int npub) :
        cPriMem_1(cPri), priMem_1(npri), cPubMem_1(cPub), pubMem_1(npub) {}
    int getCPri() { return cPriMem_1; }
    void setcPri(int i) { cPriMem_1 = i; } // Error 1: Assignment to constant data member
    int getPub() { return priMem_1; }
    void setPub(int i) { priMem_1 = i; }
}
int main() {
    MyClass myObj(1, 2, 3, 4);
    cout << myObj.getCPri() << endl; myObj.setcPri(6);
    cout << myObj.getPub() << endl; myObj.setPub(5);
    cout << myObj.cPubMem_1 << endl; // Error 2: Assignment to constant data member
    cout << myObj.pubMem_1 << endl; myObj.pubMem_1 = 5;
    return 0;
}

```

- It is not allowed to make changes to constant data members in myObj
- Error 1-value specifies const object
- Error 2-'myObj' you cannot assign to a variable that is const

NPTEL MOOCs Programming in C++ Partha Pratim Das 10

So, let's look at this. Now, we have an example where we have two private members, this is a non constant one and this is a constant one, we have made it constant by putting this const keyword. Similarly, I have shown two public data members, one of them is constant, and the other one is non constant. We have constructors which initialize all of them and we have the set get on all of these members.

Naturally, what will happen is if I am saying that my this data member is constant I do

not expect it to change. So, when we try to write a set function on this data member which will assign a value to cprimem, then the compiler gives you an error because if this could compile without any error then a an object would be able to invoke this and make changes to the data member. Be careful to understand that, in this case the member function does not need to be a const member function. This is a non const member function, but the reason you will get an error is because the member itself, the data member itself that you want to change has been declared to be constant. So, the set function cannot be invoked.

Similarly, if you look into this line where we are trying to assign a value to the public data member which has been declared as constant, we will get another error from the compiler because if you look into the declaration of the public data member, the public data member has been declared as constant. So, it should not be changable. So, this is how by using const-ness in front of the data members you can selectively make the data members constant, whereas we have been able to make changes, will be able to, for example, here we are making assignment to this particular data member, which is the public member and that assignment does not give you an error, we are changing this which is also allowed because this is not a constant member.

(Refer Slide Time: 18:56)

The slide is titled "Credit Card Example". On the left, there is a sidebar with a navigation menu:

- Module 15
- Partha Pratim Das
- Objectives & Outline
- Constant Objects
- Constant Member Functions
- Constant Data Members
- Credit Card Examples
- mutable Members
- Summary

Below the menu is a circular profile picture of a man with glasses.

The main content area contains the following text:

We now illustrate constant data members with a complete example of CreditCard class with the following supporting classes:

- String class
- Date class
- Name class
- Address class

At the bottom of the slide, there is footer information:

NPTEL MOOCs Programming in C++ Partha Pratim Das 11

So, now let us take a re look in our credit card example and see how constant data member can be used for a better design of the credit card class.

(Refer Slide Time: 19:12)

Module 15
Partha Pratim Das

Objectives & Outline
Constant Objects
Constant Member Functions
Constant Data Members
Credit Card Example
mutable Members
more

Program 15.05: String Class:
In header file with copy

```
#ifndef __STRING_H
#define __STRING_H
#include <iostream>
#include <string>
using namespace std;

class String {
    char *str;
    size_t len;
public:
    String(const char *s) : str(strdup(s)), len(strlen(str)) {} // ctor
    String() : str(nullptr), len(0) {} // default ctor
    String(const String& s) : str(strdup(s.str)), len(strlen(str)) {} // copy ctor
    String& operator=(const String& s) {
        if (this != &s) {
            free(str);
            str = strdup(s.str);
            len = s.len;
        }
        return *this;
    }
    ~String() { cout << "String dtor: " ; print(); cout << endl; free(str); } // dtor
    void print() const { cout << str; }
};

#endif // __STRING_H

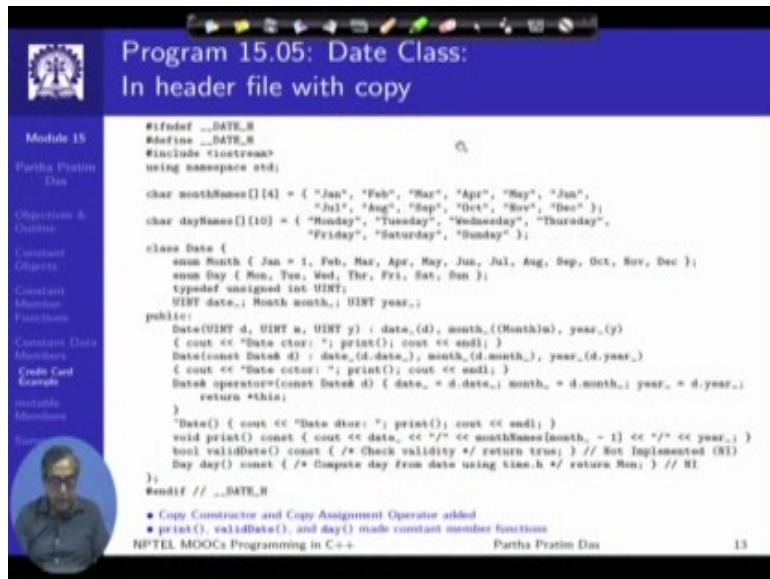
```

• Copy Constructor and Copy Assignment Operator added
• print() made a constant member function

NPTEL MOOCs Programming in C++ Partha Pratim Das 12

You have already seen all these class designs all these codes. So, I will not reiterate them, just note the small changes that have been made, for example, if you look into this string class then we have introduced here. we have introduced the copy constructor and copy assignment operator, it was discussed in a module 14 and importantly in the print function, we have introduced a const because as you said, the print function is not expected to change any data member and we have just learnt that any member function that does not change the data members must be declared as a constant member function to protect that no constant object inadvertently can change this. So, we will make all this print functions as const.

(Refer Slide Time: 20:13)



```
#ifndef __DATE_H
#define __DATE_H
#include <iostream>
using namespace std;

char monthName[] [4] = { "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };
char dayName[] [10] = { "Monday", "Tuesday", "Wednesday", "Thursday",
                       "Friday", "Saturday", "Sunday" };

class Date {
    enum Month { Jan = 1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec };
    enum Day { Mon, Tue, Wed, Thu, Fri, Sat, Sun };
    typedef unsigned int UINT;
    UINT date_, Month month_, UINT year_;
public:
    Date(UINT d, UINT m, UINT y) : date_(d), month_(Month(m)), year_(y)
    { cout << "Date ctor: "; print(); cout << endl; }
    Date(const Date& d) : date_(d.date_), month_(d.month_), year_(d.year_)
    { cout << "Date copy: "; print(); cout << endl; }
    Date operator=(const Date& d) { date_ = d.date_; month_ = d.month_; year_ = d.year_; return *this; }

    ~Date() { cout << "Date dtor: "; print(); cout << endl; }
    void print() const { cout << date_ << "/" << monthName[month_-1] << "/" << year_; }
    bool validate() const { /* Check validity */ return true; } // Not Implemented (8)
    Day day() const { /* Compute day from date using time.h */ return Mon; } // RI
};

#endif // __DATE_H

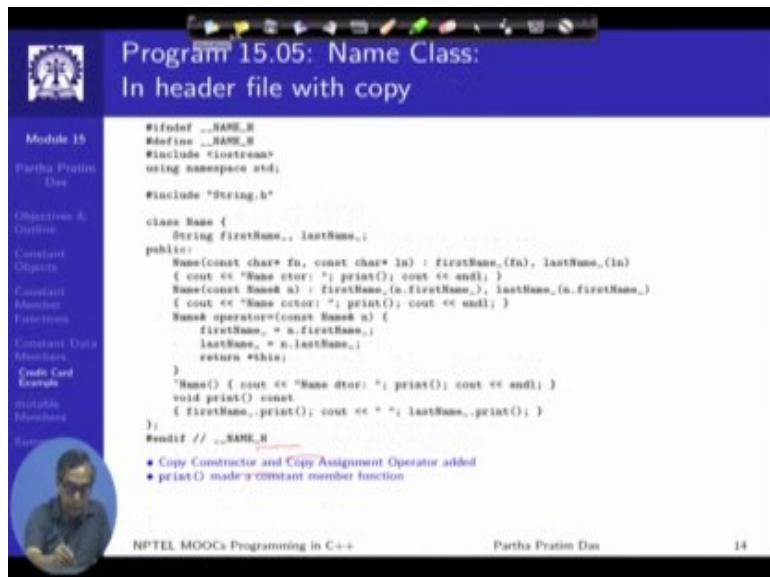
```

• Copy Constructor and Copy Assignment Operator added
• print(), validate(), and day() made constant member functions

NPTEL MOOCs Programming in C++ Partha Pratim Das 13

The date class; again in the date class we have provided the provisions for copy and we had met all this functions print, validate date, day, all these functions as const member functions as is protocol that we have agreed to.

(Refer Slide Time: 20:40)



```
#ifndef __NAME_H
#define __NAME_H
#include <iostream>
using namespace std;

#include "String.h"

class Name {
    String firstName_, lastName_;
public:
    Name(const char* fn, const char* ln) : firstName_(fn), lastName_(ln)
    { cout << "Name ctor: "; print(); cout << endl; }
    Name(const Name& n) : firstName_(n.firstName_), lastName_(n.lastName_)
    { cout << "Name copy: "; print(); cout << endl; }
    Name operator=(const Name& n)
    {
        firstName_ = n.firstName_;
        lastName_ = n.lastName_;
        return *this;
    }
    ~Name() { cout << "Name dtor: "; print(); cout << endl; }
    void print() const { firstName_.print(); cout << " "; lastName_.print(); }
};

#endif // __NAME_H

```

• Copy Constructor and Copy Assignment Operator added
• print() made a constant member function

NPTEL MOOCs Programming in C++ Partha Pratim Das 14

The name class again copies are added and the print function has been made into a

constant member function. We have seen this class also earlier it can be used to define the name of any person.

(Refer Slide Time: 21:05)

```
#ifndef __ADDRESS_H
#define __ADDRESS_H
#include <iostream>
using namespace std;
#include "String.h"

class Address {
    unsigned int houseNo_, street_(on), city_(on), pin_(pin);
public:
    Address(unsigned int hn, const char* sn, const char* on, const char* pn) :
        houseNo_(hn), street_(on), city_(on), pin_(pin)
    { cout << "Address ctor: "; print(); cout << endl; }
    Address(const Address& a) :
        houseNo_(a.houseNo_), street_(a.street_), city_(a.city_), pin_(a.pin_);
    Address& operator=(const Address& a) {
        houseNo_ = a.houseNo_; street_ = a.street_; city_ = a.city_; pin_ = a.pin_;
        return *this;
    }
    ~Address() { cout << "Address dtor: "; print(); cout << endl; }
    void print() const {
        cout << houseNo_ << " "; street_.print(); cout << " ";
        city_.print(); cout << " "; pin_.print();
    }
};

#endif // __ADDRESS_H
● Copy Constructor and Copy Assignment Operator added
● print() made a constant member function
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 15

The address class copy is added and I get the print function has been made a constant member function.

(Refer Slide Time: 21:14)

```
#ifndef __CREDIT_CARD_H
#define __CREDIT_CARD_H
#include <iostream>
using namespace std;
#include "Date.h"
#include "Name.h"
#include "Address.h"
class CreditCard { typedef unsigned int UIINT; char *cardNumber_;
    Name holder_; Address addr_; Date issueDate_, expiryDate_, UIINT cvv_; public:
    CreditCard(const char* cNumber, const char* fn, const char* ln,
        unsigned int hn, const char* sn, const char* on, const char* pn,
        UIINT issueMonth, UIINT issueYear, UIINT expiryMonth, UIINT expiryYear, UIINT cvv) :
        holder_(fn, ln), addr_(hn, sn, on, pn), issueDate_(t, issueMonth, issueYear),
        expiryDate_(t, expiryMonth, expiryYear), cvv_(cvv)
    { cardNumber_ = new char[strlen(cNumber) + 1]; strcpy(cardNumber_, cNumber);
        cout << "00 ctor: "; print(); cout << endl; }
    ~CreditCard() { cout << "00 dtor: "; print(); cout << endl; }

    void setHolder(const Name& h) { holder_ = h; } // Change holder name
    void setAddress(const Address& a) { addr_ = a; } // Change address
    void setIssueDate(const Date& d) { issueDate_ = d; } // Change issue date
    void setExpiryDate(const Date& d) { expiryDate_ = d; } // Change expiry date
    void setCvv(UIINT v) { cvv_ = v; } // Change cvv number
    void print() const { cout << cardNumber_ << " "; holder_.print(); cout << " "; addr_.print();
        cout << " "; issueDate_.print(); cout << " "; expiryDate_.print(); cout << " "; cout << cvv_ }

};

#endif // __CREDIT_CARD_H
● Set methods added
● print() made a constant member function
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 15

Now, finally, we are with the credit cards class which makes use of this string date, name, address, all these classes and we have the same set of data members and functionality. Now, what we do, we introduce a number of set methods. Earlier we have just been constructing the credit card object and printing it.

Now, we have introduced a number of set methods by which the different data members of the credit cards object can be changed that is I can set the name of a new holder, I can set the address, I can set the issue date and so on and of course, print has been made constant. So, this is the change that we have made and with this let us see what consequences happen.

(Refer Slide Time: 22:06)

```
#include <iostream>
using namespace std;

#include "CreditCard.h"

int main() {
    CreditCard cc("989171193466400927", "Sherlock", "Holmes",
                 221, "Baker Street", "London", "NW1 6XE", 7, 2014, 6, 2016, 811);
    cout << endl; cc.print(); cout << endl << endl;

    cc.setHolder("David", "Cameron");
    cc.setAddress(Address(10, "Downing Street", "London", "SW1A 2AA"));
    cc.setIssueDate(Date(1, 7, 2017));
    cc.setExpiryDate(Date(1, 6, 2019));
    cc.setCVV(123);
    cout << endl; cc.print(); cout << endl << endl;

    return 0;
}

// Construction of Data Members & Object
632171193466400927 Sherlock Holmes 221 Baker Street London NW1 6XE 1/Jul/2014 1/Jun/2016 811
// Construction & Destruction of temporary objects
632171193466400927 David Cameron 10 Downing Street London SW1A 2AA 1/Jul/2017 1/Jun/2019 127
// Destruction of Data Members & Object
■ We could change address, issue date, expiry date, and cvv. This is fine
■ We could change the name of the holder! This should not be allowed
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 17

So, if you look into the credit card application. Now, in the application we have used the set members to make changes. So, all of these changes show up in terms of it was originally created, the object was created for Sherlock Holmes having certain address and some certain dates and now, it has been edited to have a different holder name with different address and so on.

So, we could change all of these different fields which are fine, but the disturbing part is we are able to change the name of the holder which according to a credit card issue

system should not be possible. A card issue to some one cannot be re issue to someone else. So, we have to stop this possibility of changing the name of the holder. So, this is where we will now use constant data member's tool.

(Refer Slide Time: 23:15)

```

Module 15
Partha Pratim Das
Objectives & Outline
Constant Objects
Constant Member Functions
Constant Data Members
Credit Card Example
mutable Members
Summary
#ifndef __CREDIT_CARD_H
#define __CREDIT_CARD_H
// Include <iostream>, <string.h>, <Date.h>, <Name.h>, <Address.h>
using namespace std;
class CreditCard { typedefd unsigned int UERT;
    char *cardNumber;
    const Name holder; // Holder name cannot be changed after construction
    Address addr;
    Date issueDate, expiryDate; UERT cvv;
public:
    CreditCard(...); ...
    ~CreditCard() { ... }

    void setHolder(const Name& h) { holder = h; } // Change holder name
    // Error C2679: binary '=' : no operator found which takes a left-hand operand
    // of type 'const Name' (or there is no acceptable conversion)

    void setAddress(const Address& a) { addr = a; } // Change address
    void setIssueDate(const Date& d) { issueDate = d; } // Change issue date
    void setExpiryDate(const Date& d) { expiryDate = d; } // Change expiry date
    void setCvv(UERT v) { cvv = v; } // Change cvv number

    void print();
};

#endif // __CREDIT_CARD_H

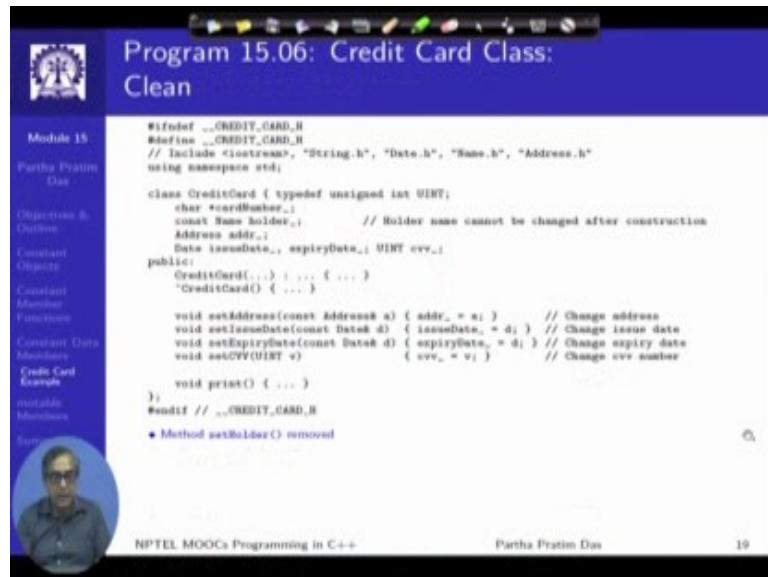
```

- We prefix `Name` with `const`. Now the holder name cannot be changed after construction
- In `setHolder()`, we get a compilation error for `holder = h;` as `h` is an attempt to change `holder..`
- With `const` prefix `Name` becomes constant = unchangeable

NITTEL MOOCs Programming in C++ Partha Pratim Das 18

So, what we do now, we have learnt how to do this. If we do not want the name of the holder to be changed, before the declaration of the name of the holder we put a `const`. So, the name of the holder now becomes a constant data member. So, in a credit object this cannot be edited which means that the set holder function that we had written the set holder function now becomes error because it is trying to assign to the holder which defines this constant. So, you get this some kind of compiler errors saying that you cannot have this kind of a function. If I cannot have this kind of a function then naturally I cannot change the name of the holder. So, we achieve our objective in that.

(Refer Slide Time: 24:09)



```
#ifndef __CREDIT_CARD_H
#define __CREDIT_CARD_H
// Include <iostream>, <string.h>, <Date.h>, <Name.h>, <Address.h>
using namespace std;

class CreditCard { typedef unsigned int UINT;
    char *cardHolder_; // Holder name cannot be changed after construction
    const Name holder_; // Holder name cannot be changed after construction
    Address addr_; // Holder address cannot be changed after construction
    Date issueDate_, expiryDate_; // Expiry date cannot be changed after construction
    UINT cvv_ = 0; // CVV number cannot be changed after construction

public:
    CreditCard(...); // ...
    ~CreditCard(); // ...

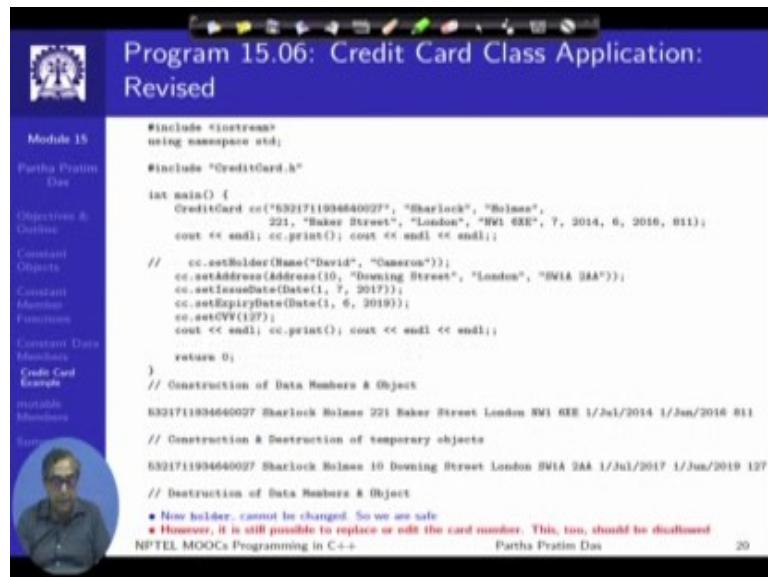
    void setAddress(Address a) { addr_ = a; } // Change address
    void setIssueDate(Date d) { issueDate_ = d; } // Change issue date
    void setExpiryDate(Date d) { expiryDate_ = d; } // Change expiry date
    void setCvv(UINT v) { cvv_ = v; } // Change cvv number

    void print();
};

#endif // __CREDIT_CARD_H
* Method setHolder() removed
```

So, this is just the same class cleaned up. We have now removed the set holder function because that function cannot be compiled and therefore, there is no way to change the set holder and we will use this too.

(Refer Slide Time: 24:25)



```
#include <iostream>
using namespace std;

#include "CreditCard.h"

int main() {
    CreditCard cc("5321711934640027", "Sherlock", "Holmes",
        221, "Baker Street", "London", "SW1 6EE", 7, 2014, 6, 2016, 811);
    cout << endl; cc.print(); cout << endl << endl;

    cc.setHolder("David", "Cameron");
    cc.setAddress(Address(10, "Downing Street", "London", "SW1A 2AA"));
    cc.setIssueDate(Date(1, 7, 2017));
    cc.setExpiryDate(Date(1, 6, 2019));
    cc.setCvv(123);
    cout << endl; cc.print(); cout << endl << endl;

    return 0;
} // Construction of Data Members & Object
5321711934640027 Sherlock Holmes 221 Baker Street London SW1 6EE 1/Jul/2014 1/Jun/2016 811
// Construction & Destruction of temporary objects
5321711934640027 Sherlock Holmes 10 Downing Street London SW1A 2AA 1/Jul/2017 1/Jun/2019 127
// Destruction of Data Members & Object
* Now holder cannot be changed. So we are safe
* However, it is still possible to replace or edit the card number. This, too, should be disallowed
```

Again go through the application once more. Now, in the application, we have this is

commented out this cannot be done, but rest of the changes can still be done. They can still change the address, the issue date and all those. So, we are safe, but it is still possible that I can actually change the card number itself. I can edit the card number or replace the card number, put a different card number. So, how do we guard against that? How do we take care that this cannot be changed?

(Refer Slide Time: 25:02)

```

Module 15
Partha Pratim Das
Objectives & Outline
Constant Objects
Constant Member Functions
Constant Data Members
Credit Card Example
mutable Members
Summary

Program 15.07: Credit Card Class:
cardMember_Issue

#ifndef __CREDIT_CARD_H
#define __CREDIT_CARD_H
// Include <iostream>, <string.h>, <Date.h>, <Name.h>, <Address.h>
using namespace std;

class CreditCard { typedef unsigned int UERT;
    char *cardNumber_; // Card number is editable as well as replaceable
    const Name holder_; // Holder name cannot be changed after construction
    Address addr_;
    Date issueDate_, expiryDate_; UERT cvv_;
public:
    CreditCard(...); ...
    ~CreditCard() { ... }

    void setAddress(Address a) { addr_ = a; } // Change address
    void setIssueDate(Date d) { issueDate_ = d; } // Change issue date
    void setExpiryDate(Date d) { expiryDate_ = d; } // Change expiry date
    void setCvv(UERT v) { cvv_ = v; } // Change cvv number

    void print(); ...
};

#endif // __CREDIT_CARD_H



- ★ It is still possible to replace or edit the card number
- ★ To make the cardholder non-replaceable, we need to make this pointer constant
- ★ Further, to make it non-editable we need to make cardholder point to a constant string
- ★ Hence, we change char *cardholder to const char * const cardholder.

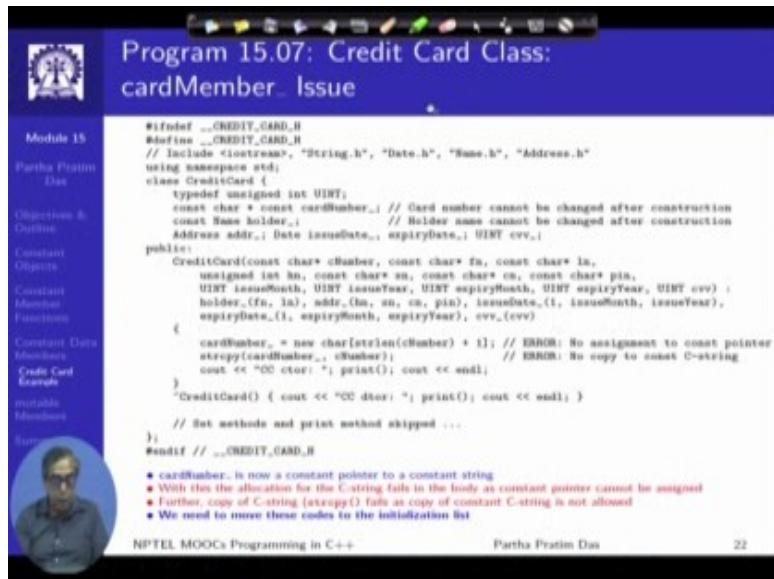
```

NPTEL MOOCs Programming in C+++ Partha Pratim Das 23

For this we will have to see that a card number is a pointer variable. So, it is supposed to get allocated and it will point to a string. So, there are two ways that the card number can get affected, one is I can traverse this pointer and change the string, I can edit the object itself.

The other is I could reallocate a new object and put in the place of the card number there is I can just reallocate this. So, this is the situation like this is a card number, there is an allocation here of the card number some 4, 3, 8, 6 and so on. So, I can either edit this or I can change, add a new card number. so both of this have to be stopped. This should not be possible and we discuss this situation where we discussed about const-ness of pointers that here what we need is we need the pointer as well as the pointed object to be both to be constant.

(Refer Slide Time: 26:15)



```
#ifndef __CREDIT_CARD_H
#define __CREDIT_CARD_H
// Include <iostream>, "String.h", "Date.h", "Name.h", "Address.h"
using namespace std;
class CreditCard {
    typedef unsigned int UINT;
    const char * const cardNumber_; // Card number cannot be changed after construction
    const Name holder_; // Holder name cannot be changed after construction
    Address addr_; Date issuance_, expiryDate_; UINT cvv_;
public:
    CreditCard(const char* cNumber, const char* fn, const char* ln,
               unsigned int bn, const char* sn, const char* cn, const char* pin,
               UINT issueMonth, UINT issueYear, UINT expiryMonth, UINT expiryYear, UINT cvv) :
        holder_(fn, ln), addr_(bn, sn, cn, pin), issuance_(t, issueMonth, issueYear),
        expiryDate_(t, expiryMonth, expiryYear), cvv_(cvv)
    {
        cardNumber_ = new char[strlen(cNumber) + 1]; // ERROR: No assignment to const pointer
        strcpy(cardNumber_, cNumber); // ERROR: No copy to const C-string
        cout << "CC ctor: "; print(); cout << endl;
    }
    ~CreditCard() { cout << "CC dtor: "; print(); cout << endl; }

    // Set methods and print method skipped ...
};

#endif // __CREDIT_CARD_H
```

* cardNumber_ is now a constant pointer to a constant string
* With this the allocation for the C-string fails in the body as constant pointer cannot be assigned
* Further, copy of C-string (strcpy) fails as copy of constant C-string is not allowed
* We need to move these codes to the initialization list

So, this is what we simply do, we make it a constant pointer and we make it point to constant string constant object and with this it should not be possible, it would not be possible to change it, but now it face a different kind of problem. The problem is look at how this was initialized in the body of the constructor, you know at this point the object has been constructed, initialization is over and in this we have not initialized the card member because there is quite some code to write, to initialize that. you have to allocate the string which we do here.

Before that we have to get the length of the string then we allocate the string and then we copy the given numbers strings in to this. So, we decided to do this in the body of the constructor, but now that this has become a constant pointer this assignment is not possible, because the card member has already been initialized in the initialization list further since, the object itself is constant. This copy is not possible because the object also is constant, it is pointing to a constant object. So, if I do strcpy, if I copy the string then I am actually changing that string object. So, both of these will give an error. So, we will have to make sure, we will have to take care that we actually do all of these task in the initialization list itself.

(Refer Slide Time: 27:45)

```
#include <iostream>
using namespace std;
#include "String.h"
#include "Date.h"
#include "Name.h"
#include "Address.h"
class CreditCard {
    typedef unsigned int UINT;
    const char * const cardNumber_; // Card number cannot be changed after construction
    const Name holder_; // Holder name cannot be changed after construction
    Address addr_; Date issueDate_, expiryDate_; UINT cvv_;
public:
    CreditCard(const char* cNumber) : const char* fn, const char* ln,
        unsigned int bn, const char* sn, const char* cn, const char* pin,
        UINT issueMonth, UINT issueYear, UINT expiryMonth, UINT expiryYear, UINT cvv) :
        cardNumber_(strcpy(new char[strlen(cNumber)+1], &cardNumber)),
        holder_(fn, ln), addr_(bn, sn, cn, pin), (issueDate_(11, issueMonth, issueYear),
        expiryDate_(1, expiryMonth, expiryYear)), cvv_(cvv)
    {
        cout << "Card created" << endl;
    }
    ~CreditCard() { cout << "Card destroyed" << endl; }
    void setAddress(Address a) { addr_ = a; } // Change address
    void setIssueDate(Date d) { issueDate_ = d; } // Change issue date
    void setExpiryDate(Date d) { expiryDate_ = d; } // Change expiry date
    void setCvv(UINT v) { cvv_ = v; } // Change cvv number
    void print() { cout << cardNumber_ << endl; holder_.print(); cout << " "; addr_.print();
        cout << " "; issueDate_.print(); cout << " "; expiryDate_.print(); cout << " "; cout << cvv_ << endl; }
};
```

1641 -> 17

- Note the initialization of cardNumber_ in initialization list
- All constant data members must be initialized in initialization list

We will not able to do this in the body of the constructor. Now, if you look carefully, we have introduced the initialization of the card member in here. Please try to understand this, do not get scared by the little complexity of this expression, if you go one by one C number is basically the given card string. So, first we do find the length of it then we make an allocation for it. So, we get a pointer having adequate space where the card number can be put and then we do s t r c p y to copy C number into this location. So, it is like first we get the length. So, the length is 16, we add 1; so we get 17.

This 1 is for the null character then you make an allocation. So, we have an allocation of an array of 16 into some location and then we copy whatever my input string is into this using the s t r c p y. So, that my card member gets initialized to this string and it will have to be done here it will have to be done in the initialization list because both the pointer card MEM number pointer as well as the string that it will point to have become constant now, but with this we are able to protect and construct the object in a way, so that you cannot make any changes to the card number, once the credit card object has got created.

(Refer Slide Time: 29:49)

The screenshot shows a presentation slide titled "mutable Data Members". The slide content is as follows:

- While a **constant** data member is *not changeable even in a non-constant object*, a **mutable** data member is *changeable in a constant object*
- mutable** is provided to model *Logical (Semantic) const-ness* against the default *Bit-wise (Syntactic) const-ness* of C++.
- Note that:
 - mutable** is applicable only to data members and not to variables
 - Reference data members cannot be declared **mutable**
 - Static data members cannot be declared **mutable**
 - const** data members cannot be declared **mutable**
- If a data member is declared **mutable**, then it is legal to assign a value to it from a **const** member function
- Let us see an example

The slide footer includes the text "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" along with a page number "24".

We have just seen how to create constant objects, constant member functions and constant data members and how to use that in the design.

Programming in C++
Prof. Partha Pratim Das
Department Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 30
Const-ness (Contd.)

Welcome back to Module 15 of Programming in C++. We have been discussing Const-ness, we have introduced constant objects; we have seen how the this pointer of a constant object changes in its type because now it becomes a constant pointer pointing to a constant object.

We have seen with a constant object we cannot invoke a normal or non constant member function and we introduced the notion of constant member function which has a this pointer of the same type, that is a constant pointer to a constant object and we have seen with the use of constant member function, how we can protect the constant object and still invoke member function to read or access the value of the data members and we have learnt that whenever in a class, a member function is not changing any object then it should be made has const.

Of course, if a member function wants to change a part of the object, any one of the data members also then it cannot be a const member function and complier will give you an error; obviously, those member function cannot be invoked from const objects.

We have also seen that part of the object can be selectively made constant by using const data member and we have seen the const data members cannot be changed by any of the member functions and they have to be initialized, they must be initialized at the time of construction and with the credit card example we have shown how the const data members can be used to create class designs where the known editable data, whether they are normal data or they are pointed type data can be protected from the any changes to be done from the application.

Now we will proceed and talk about another feature, another selective const-ness feature known as Mutable Data Members.

(Refer Slide Time: 02:35)

The screenshot shows a presentation slide titled "mutable Data Members". The slide content is as follows:

- While a *constant* data member is *not changeable* even in a *non-constant* object, a **mutable** data member is *changeable* in a *constant* object.
- mutable** is provided to model *Logical (Semantic) const-ness* against the default *Bit-wise (Syntactic) const-ness* of C++.
- Note that:
 - mutable** is applicable only to data members and not to variables.
 - Reference data members cannot be declared **mutable**.
 - Static data members cannot be declared **mutable**.
 - const* data members cannot be declared **mutable**.
- If a data member is declared **mutable**, then it is legal to assign a value to it from a *const* member function.
- Let us see an example.

The slide has a navigation bar on the left with the following items: Module 15, Partha Pratim Das, Objectives & Outline, Constant Objects, Constant Member Functions, Constant Data Members, Code and Examples, mutable Members, and Summary. At the bottom, it says NPTEL MOOCs Programming in C++ and Partha Pratim Das, with a page number 24.

Mutable - concept of being mutable is closely related to the concept of const-ness. As we know that a constant data member is not changeable, even when the object is non-constant, this is what we have learnt. On the other hand, a mutable data member is changeable in a constant object. So, a constant object as such is not changeable, nothing in that constant object can be changed, but if I define a data member of a class as mutable then even when an object of that class is defined to be constant even in that case that particular data member can be changed and that is the genesis of this name mutable as you know mutation means change, so mutable means changeable data member.

Now, there is a significant reason and specific purpose of why mutable data members exist. Mutable key word is provided or the mutable feature is provided to support logical const-ness against the bitwise const-ness of C++, what I mean is if I declare an object to be *const*; I say this object is constant then I am saying that nothing can be changed in that. If nothing can be changed in that; that means, whatever bit pattern that particular object has got at the time of initialization, no changes to that bit pattern ever is possible, but that often becomes little bit difficult to work with, I may have a concept which is constant, but its implementation may need additional fields, additional parameters, additional data members which are not exactly constant, so we will look into that.

Meanwhile you note, please note that mutable is applicable only in the case of data members, if you just have another a simple variable you cannot declare it to mutable and any reference data static data member of course you have not done static data member yet, we will talk about next module or constant data members can be declared as mutable. The most important part is if a data member is declared as mutable then it is legal to change it in a constant member function. We have seen that a constant member function cannot change any data member, but if a data member is mutable then it can still change it, because you are saying mutable means that it can be changed even when things are constant. So, let us look at to an example.

(Refer Slide Time: 05:37)

```
#include <iostream>
using namespace std;
class MyClass {
    int mm_1;
    mutable int mutableMem_;
public:
    MyClass(int m) : mm_(m), mutableMem_(m) {}
    int getMem() const { return mm_1; }
    void setMem(int i) { mm_1 = i; }
    int getMutableMem() const { return mutableMem_; }
    void setMutableMem(int i) const { mutableMem_ = i; } // Okay to change mutable
};

int main() {
    const MyClass myConstObj(1, 2);
    cout << myConstObj.getMem() << endl;
    //myConstObj.setMem(3);           // Error to invoke
    cout << myConstObj.getMutableMem() << endl;
    myConstObj.setMutableMem(4);
    return 0;
}

* setMutableMem() is a constant member function so that constant myConstObj can invoke it
* setMutableMem() can still set mutableMem_, because mutableMem_ is mutable
* In contrast, myConstObj cannot invoke setMem() and hence mm_1 cannot be changed

```

So, here I have introduced one data member which is mutable, so you write it syntax similar to writing const in place of const you are just writing mutable and then we have declared a constant object, now we have a get method, we have a set method. Now we are; the get set on the non mutable one are different, the get is a const member function, set is a non const member function. So, which what means that if I have my const object, then I can do get on it, I cannot do set on it because constant objects cannot invoke non constant member.

Now look at the mutable part, here both these member functions had been declared to be const, so both of them can be invoked by the constant object. Now this function; set function is trying to change the member, if this particular member were not mutable then

this would have been a compilation error, this would not have been possible, but since this member, data member is mutable it is allowed to change it. Therefore, even though this is a const member function, it can make changes to the data member, that is the basic consequence of having a mutable data member, so this is the basic behavior of the feature. Now let us try to see how this can be used, how this should be used.

(Refer Slide Time: 07:18)

The slide has a blue header with the title 'Logical vis-a-vis Bit-wise Const-ness'. On the left, there's a sidebar with navigation links: 'Module 15', 'Partha Pratim Das', 'Objectives & Outline', 'Constant Objects', 'Constant Member Functions', 'Constant Data Members', 'Code Examples', 'mutable Members', and 'Summary'. Below the sidebar is a circular portrait of Partha Pratim Das. The main content area contains a bulleted list:

- **const** in C++, models *bit-wise* constant. Once an object is declared **const**, no part (actually, *no bit*) of it can be changed after construction (and initialization)
- However, while programming we often need an object to be *logically* constant. That is, the concept represented by the object should be constant; but if its representation need more data members for computation and modeling, these have no reason to be constant.
- **mutable** allows such surrogate data members to be changeable in a (bit-wise) constant object to model logically const objects
- To use **mutable** we shall look for:
 - A logically constant concept
 - A need for data members outside the representation of the concept; but are needed for computation

At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

So, as I was explaining **const** in C++ models bit wise constants that nothing in the bits of the constant object can change, but what is often important is I have a concept which is logically constant, that I am trying to model some constant, concept. So, that is described in terms of a few data members, but when I want to do something computation; in that same class I need some more data members which I just supporting the computation. Now if I make the object constant since all bits are constant nothing can change certainly all those I mean so called mutable surrogate data members which are just supporting the computation they also cannot change and therefore I cannot write the code.

Mutable has been provided to work around this problem, so to be able to use or to effectively use **mutable**, we will basically; please try to remember these two point, we will try to use; look for a logically constant concept, it is just not for a programming end there must be a concept which is logically constant and to implement that concept, we should need some data members which are outside of that concept, which are just written

required for the computation, I am sure this is not sounding very clear, so let me take an example.

(Refer Slide Time: 09:01)

Module 15
Partha Pratim Das
Objectives & Outcome
Constant Objects
Constant Member Functions
Constant Data Members
Code and Examples
mutable Members

**Program 15.09:
When to use mutable Data Members?**

• Typically, when a class represents a constant concept, and
• It computes a value first time and caches the result for future use

// Source: <http://www.highprogrammer.com/alan/rants/mutable.html>

```
#include <iostream>
using namespace std;
class MathObject {
    mutable bool piCached_=false;           // Constant concept of PI
    mutable double pi_=3.14159;              // Needed for computation
public:
    MathObject() : piCached_(false) { } // Not available at construction
    double pi() const {                // Can access PI only through this method
        if (!piCached_) {             // An insanely slow way to calculate pi
            pi_ = 4;
            for (long step = 3; step < 1000000000; step += 4) {
                pi_ += ((-4.0 / (double)step) * (4.0 / ((double)step + 2)));
            }
            piCached_ = true;          // Now computed and cached
        }
        return pi_;
    }
};

int main() {
    const MathObject mo;
    cout << mo.pi() << endl; // Access PI
    return 0;
}
```

• Here a MathObject is logically constant; but we use mutable members for computation

NPTEL MOOCs Programming in C++ Partha Pratim Das 27

We will read the top notes later on; first let us understand what we are doing. We are defining a class math object and my intention of doing this is, in this sample case my intention is math object should give me a constant object which is pi, we all know pi 3.14159 and so on. So, what I want is it should whenever I will construct object construct a constant object of this class, I will not construct non constant object.

So, I will construct this and on this if I invoke this method pi, then it should give me the value of pi. Now why do we need get the value of pi, this is the algorithm say as you know pi is a transcendental number, so I cannot just put down, it is a long chain of approximation, long tail of approximation. So, I provide an algorithm which as it happens, there are better algorithms to compute pi, but this one happens to a pretty slow algorithm. So, if I invoke this mo dot pi; mo is the object, so if I do mo dot pi then this member function will be called which is a constant member function it can be called an constant object, this algorithm will execute and give me the value of pi.

Now, since this is potentially very slow, this can be potentially very slow, I want to do some caching which means see I know the pi is a constant; concept is a constant. So, if I compute it once, the second time as I compute it will give me the same value, so why should I repeatedly compute, so I want to remember that whether I have computed it or I

have not computed it. So, the first time I compute it, next time onwards I should use that same computed value, so I use two data members in this, one is the value of pi; initially that value I do not know, in the constructor I cannot put this because I do not know what the value is, it has to be computed.

I will not do that computation unless somebody wants to use the value pi because this takes a lot of time as I said and when first time pi is to be used then this member function will get called. Now, I am maintaining another data member pi cached which is a bool flag, which remembers whether pi has been computed or it has not been computed. So, that flag is initially set to false in the constructor of the object, so it says that it has not been computed.

So, now if I call pi, then this check will say the pi has been not computed, this whole computation will happen, the value will get updated in the data member pi and as I complete the iteration, I come out and put pi cached to be true. So, in future when ever this data member pi is; a member function pi is called again, this will turn out to true therefore, whole of this will bypassed and I will simply be able to; this becomes, this will behave like a get function, this will just become get one pi, so this ensures that it will just be executed once and will be used number of times.

So let us see how you could have done this, the first thing that we require, we need to say that this object has to be constant because pi cannot be a non-constant. So the concept that is constant is pi, so that this is my basic requirement, it has to be a constant. Now if this is constant then these data members cannot be changed, now of these two data members; of these data members for example, let us say the cache data member, this cached data member is not a part of the concept.

Caching the value of pi is not a constant concept the constant-ness of the concept is in the value of pi. So, the pi cashed is been required because I want to do the computation, this pi; the value itself is required to be modified because I do not know what that value is when I am initializing. So, these data members are basically supporting the implementation of the concept so if you just take that; I will have a constant object then mo; mo will have false and whatever value becomes a bit wise constant nothing can be changed, so this whole strategy will not work. So, I make these two data members as mutable (Refer Time: 14:12) even though the object is constant, the concept is constant,

but in the implementation I need some data members which need not be constant and this sense of const-ness is called the logical const-ness.

So, if you look into the bit pattern; bit pattern is changing, but what you wanted represent is a basic concept of pi being constant that is preserved. So, mutable has a very specific purpose in this manner and that is a typical case to the best of my knowledge possibly the only case where mutable should be used. So, now, you can read the initial comments, this typically used when a class represent a constant concept and it computes a value first time and caches the result for future use.

(Refer Slide Time: 15:06)

Module 15

Partha Pratim Das
Objectives & Outline
Constant Objects
Constant Member Functions
Constant Data Members
Code Examples
mutable Members
Summary

Program 15.10: When not to use mutable Data Members?

* mutable should be rarely used – only when it is really needed. A bad example follows:

Improper Design (mutable)	Proper Design (const)
<pre>class Employee { string _name; string _id; mutable double _salary; public: Employee(string name = "No Name", string id = "000-00-0000", double salary = 0) : _name(name), _id(id) { _salary = salary; } string getName() const; void setName(string name); string getId() const; void setId(string id); double getSalary() const; void setSalary(double salary); void promote(double salary) const { _salary = salary; } }; const Employee john("JOHN", "007", 6000.0); // ... john.promote(20000.0);</pre>	<pre>class Employee { const string _name; const string _id; double _salary; public: Employee(string name = "No Name", string id = "000-00-0000", double salary = 0) : _name(name), _id(id) { _salary = salary; } string getName() const; string getId() const; double getSalary() const; void setSalary(double salary); void promote(double salary) { _salary = salary; } }; Employee john("JOHN", "007", 6000.0); // ... john.promote(20000.0);</pre>

* Employee is not logically constant. If it is, then _salary should also be const
* Design on right makes that explicit

NPTEL MOOCs Programming in C++ Partha Pratim Das 28

So, just to complete the discussion on mutable, I do present an example to show when not to use mutable, because you know it is a very symmetric concept. I can make data members constant in a non constant object and I can make data members non constant in a constant object are very symmetric concept. So, I have often seen that students tend to make the mistake of using them as equivalent of complementary concept, but that is not true.

So, to show I just show an example of an employee class, the employee has name, ID and salary. So, I say that okay, employees once created let us say this is an organization where no changes of employees are possible, so employees once created are constant objects, they cannot be changed. So, in the application we write employee is constant, but the salary of the employees change, employees need to be promoted, they need to be

promoted then they will get a better salary and if the employee is constant, if John is constant then this is not possible, so I make it a constant member function.

If this is a constant member function then this itself is an error because constant member function cannot change data member; so to work around I put a mutable, this code will be compile, this code will run, this code will give result, there is nothing wrong, so far as the syntax of C++ is concerned but so far as the design, the concept is involved or so far as what others will understand from that code is involved, this is a disaster. This is a disaster because the mutable has been introduced to particularly model constant concepts, employees cannot be constant concepts, if they are constant concepts then their salary will also be constant, the salary should also not be changeable because that is a part and parcel of the employee.

So, rather this should be model in the way we had shown earlier that make the employee objects non constant because they do not need to be constant; they are not constant by concept but just make those data members constant which cannot change for an employee that is the name and the ID and make the changeable data member as a non-constant data member so that you can normally invoke promote on this, promote now does not need to be a constant member function and therefore can make changes to the salary as required.

So, if you carefully study the design of these two parts; the associated part and the associated use, code wise both of them will work, but concept wise this is an improper design, this should not be done and this is the right way of doing a normal class design, only when you have constant concepts that you want to represent whether be it natural constants or other kinds of constant concepts then you should use mutable for making your implementation type data members as editable.

(Refer Slide Time: 18:40)

Module Summary

- Studied const-ness in C++
- In C++, there are three forms of const-ness
 - Constant Objects:
 - No change is allowed after construction
 - Cannot invoke normal member functions
 - Constant Member Functions:
 - Can be invoked by constant (as well as non-constant) objects
 - Cannot make changes to the object
 - Constant Data Members:
 - No change is allowed after construction
 - Must be initialized in the initialization list
- Further, learnt how to model logical const-ness over bit-wise const-ness by proper use of mutable members

NPTEL MOOCs Programming in C++ Partha Pratim Das 29

So, with this we come to the end of the current module. Here we have studied const-ness in C++, in C++ we have seen three forms of const-ness, the object as a whole can be constant and if the object is constant, then it can only invoke constant member functions.

So, we have seen that a constant member functions cannot change the object, but non constant objects can also invoke constant member functions and if we want to selectively make a part of the object constant like the ID of an employee, roll number of a student; then we can make the corresponding data member constant, then constant member function or non constant member function, none of them can change the constant data member. We have also seen that C++ by default supports bit wise const-ness in the const use, but it is possible to use mutable data member to achieve logical const-ness that we can have a logically constant const-ness concept which we can code in C++ using the mutable data member.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 31
Static Members

Welcome to Module 16 of Programming in C++. In this module, we will talk about static members; specifically we will try to understand the static data members and static member function.

(Refer Slide Time: 00:38)

The screenshot shows a presentation slide titled "Module Outline". On the left, there is a vertical sidebar with a logo at the top and a list of "Module 16" topics. The main content area displays a bulleted list of topics under "Objectives & Outline". At the bottom right, there is a video player interface showing a portrait of a man (Partha Pratim Das) and some playback controls.

Module 16

Partha Pratim Das

Objectives & Outline

- static data member
 - Print Task
- static member function
 - Print Task
- Singleton Class

NPTEL MOOCs Programming in C++ Partha Pratim Das

The outline will comprise these two kinds of members, and we will have a discussion about singleton classes. The outline will be visible on the left of your every slide.

(Refer Slide Time: 00:54)

The screenshot shows a presentation slide titled "static Data Member". The slide content is as follows:

A static data member

- is associated with class not with object
- is shared by all the objects of a class
- needs to be defined outside the class scope (in addition to the declaration within the class scope) to avoid linker error
- must be initialized in a source file
- is constructed before main() starts and destructed after main() ends
- can be private / public type
- can be accessed
 - with the class-name followed by the scope resolution operator (::)
 - as a member of any object of the class
- virtually eliminates any need for global variables in OOPs environment

NPTEL MOOCs Programming in C++ Partha Pratim Das

Partha Pratim Das

A static data member, we have learnt about data members. So, a data member is a component of a class structure. So, whenever a class is instantiated in terms of an object, the object will have the data members, and those data members we know can be accessed from the pointer of the object or from within the member functions of the object.

In terms of the static data member, the things are very different. Static data members are not associated with the object; like the normal data members are associated with the object, but the static data members are in contrast are associated with the class of which the array member. So, the critical property of static data member is that no matter how many instances of a class is created there will be only one instance of the static data member of a class. So, it is shared by all objects of the class.

In fact, the interesting fact that we will see over the different examples is that a static data member may exist, will exist even when no object has been created for the class. So, the purpose of the static member is primarily to maintain information to maintain data that is specific to the class, but not specific to each and every instance.

And we will see there are several programming situations, there are several problems for which having such a static data member would really help. For those of you who have

some familiarity with java, the static data member is a class variable in java, in contrast to the instance variable which is the ordinary non-static data member in C++.

For the static data member, we will see some additional constraints; we will see that the static data member is declared within the class like the ordinary data members. But it needs to be defined outside the class that is the memory binding for the static data member has to be done further outside the class typically in a source file, which can be initialized. And the interesting fact is this static member of course, is any data member could be an object. So, static data member is also an object.

The static data member gets constructed before the main function starts; this is something which probably you have not seen before in earlier cases. We always thought that when we start executing ,the main is when objects is start getting constructed, but the static members can get created or will get created before the main starts and symmetrically they will get destructed after the main ends after the main returns. So, we will see this through the example, static data members can be public or private any of the access specifier is possible.

Now in terms of accessing a static data member we will need to use the class name because as I have already mentioned here that it is associated with a class and not with the object. So, the typical object dot kind of notation is not the best way to access the static data members. They will be accessed from the class name using the colon-colon or scope resolution operator.

And in terms of usage, the static data members are nothing other than they behave exactly like the global variables. The only difference being or the significant difference being that the global variables exist in the global scope, but static data members are within the scope of the class in which the static data member belongs. So, they virtually eliminate the requirement of having any global data in the program. So, with this introduction, let me take you through a simple example and then these points will become more and more clear.

(Refer Slide Time: 05:26)

Non static Data Member

```
#include<iostream>
using namespace std;
class MyClass { int x; // Non-static public:
    void get() { x = 15; }
    void print() {
        x = x + 10;
        cout << "x=" << x << endl;
    }
};

int main() {
    MyClass obj1, obj2;
    obj1.get();
    obj2.get();

    obj1.print(); obj2.print();
    return 0;
}
```

x = 25 , x = 25

- x is a non-static data member
- x cannot be shared between obj1 & obj2
- Non-static data members do not need separate definitions - instantiated with the object
- Non-static data members are initialized during object construction

static Data Member

```
#include<iostream>
using namespace std;
class MyClass { static int x; // Declare static public:
    void get() { x = 15; }
    void print() {
        x = x + 10;
        cout << "x=" << x << endl;
    }
};

int MyClass::x = 0; // Define static data member
int main() {
    MyClass obj1, obj2;
    obj1.get(); obj2.get();

    obj1.print(); obj2.print();
    return 0;
}
```

x = 25 , x = 35

- x is static data member
- x is shared by all MyClass objects including obj1 & obj2
- static data members must be defined in the global scope
- static data members are initialized during program start-up

This is an example of a static data member; it is very simple case. So, all that I have is a class, it is some this class does not do anything very meaningful, it is called by my class and it has ordinary data member here int x. So, on the left column, what you see, does not use static member it is what you have been introduced to earlier in terms of the data members and member function. So, x is a data member, there is a get member function which initializes x with 15. And there is a print member function, which takes the current value of x, increments it by or adds 10 to that, and then prints that value. So, these are simple thing we are trying to do.

So, now if we look into the application code here, we are constructing two such objects object 1 and object 2, as you can see that my class does not have any constructor. So, we know the free default constructor will be used; and by that constructor, x will not be assigned in a specific value. So, we do invoke get on object 1. So, if we invoke get on object 1 in here, then certainly for object 1, so if I can draw in this is my object 1, this is my object 2, then I have x in both. And as I execute get for object 1, this will become 15; as I execute get for object 2, this will become 15. And then if I do print, invoke print for object one, so it will print will take this value 15 as in here add 10 to it. So, this will become 25, and therefore, it will print 25. It prints x is equal to 25.

Similarly, print is again invoked for object 2, it again takes this value which is 15 adds 10 to it, because I have called print, the value becomes 25 and this is what is printed. So, this behavior if you look at then this behavior is already very clear to you. So, you will know that I am doing exactly the same thing with both the objects object 1 and object 2, and therefore, both of them print the same value 25 for x.

Now, look at the right column, where we are trying to use the static data member. So, how do I use a static member, earlier I was saying this is int x, now I am saying it is static int x. So, I have prefixed the keyword static in front of the data member declaration. So, this is a non-static declaration, this now becomes a static declaration. Now with this the difference that that happens is if I redraw what I had on this side with a non-static data I had obj 1, obj 2, data member x, data member x, here when I will construct the objects at this line, I will have obj 1, I will have obj 2, but none of these objects obj 1 and obj 2 has x as a data member, because x is a static data member and it is not associated with any specific object.

So, there is another object whose name is MyClass :: x, my class :: x, which is the static data member x of this class. So, you can see that the moment I define this to be static my data member is not a part of the object neither a part of object 1 nor a part of object 2, but it becomes a separate location, it becomes a separate object. Now the only thing is a name of that object is not x, it is MyClass :: x that it is qualified by the class name and that is how I access it.

So, let us see when I tried to use it using the same application that I had here this there is no difference between this application and this application in main. The objects have got constructed as we have seen. I invoked get, so x is assigned 15. So, what does this mean what is x, x here is the x of MyClass, so it is this variable. So, this becomes 15. This static value becomes 15. I again execute get for invoke get for obj 2, the same x is assigned 15, because x is static it is common between two objects. So, x is set to 15 twice.

Now I invoke print for obj 1, so what will happen the value of x is 15 and 10 is added to it, so the value of x will now become 25, because 10 has got added, and that value is

what will get printed. So, when I do obj1.print x is printed as 25. Till this point, there is no difference in the behavior from the non-static case.

But consider the next one when the print is invoked for obj 2. When print is invoked for obj 2, the value of x now is 25, it is not fifteen anymore. So, this will, 10 will get added to this, and now it will become 35 and x now becomes 35. And so, when I print according to this print method then the value that I get printed is 35, because the static object is shared by the both objects. So, when I have invoked print twice, once through object 1, and once through object 2, the value 10 has been added to x twice. And therefore, earlier it was 25; in the second case now with the static data member, it becomes 35, so that is the basic behavior of static data members that we will have to learn.

(Refer Slide Time: 12:50)

```

Module 16
Partha Pratim Das
Objectives & Outline
static data member
Print Task
static member function
Print Task
Singleton Class
Summary

Non static Data Member
#ifndef MyClass_H
using namespace std;
class MyClass { int x; // Non-static
public:
    void get() { x = 15; }
    void print() {
        x = x + 10;
        cout << "x = " << x << endl;
    }
};

int main() {
    MyClass obj1, obj2;
    obj1.get(); obj2.get();

    obj1.print(); obj2.print();
    return 0;
}
x = 25 , x = 25

* x is a non-static data member
* x cannot be shared between obj1 & obj2
* Non-static data members do not need separate definitions - instantiated with the object
* Non-static data members are initialized during object construction

static Data Member
#ifndef MyClass_H
using namespace std;
class MyClass { static int x; // Declare static
public:
    void get() { x = 15; }
    void print() {
        x = x + 10;
        cout << "x = " << x << endl;
    }
};

int MyClass::x = 0; // Define static data member
int main() {
    MyClass obj1, obj2;
    obj1.get(); obj2.get();

    obj1.print(); obj2.print();
    return 0;
}
x = 25 , x = 35

* x is static data member
* x is shared by all MyClass objects including obj1 & obj2
* static data members must be defined in the global scope
* static data members are initialized during program start-up

```

So, we have noted that x is a static data member it is shared by the two objects we have seen how they are shared. And this next point is very important, let me explain this point. If you compare the two programs other than the static keyword here, there is one additional line that has been included in this. We say that this when saying that this data member is static, this is nearly a declaration, which means that it is defining it is saying that the name of this variable is x, it is a class variable. It will have one instance for the

class it is of type int, but it does not associate any memory with this variable. When it is non-static, I do not care about this, because if it is non-static I know at some point some object will get instantiated like obj 1; and wherever obj 1 gets instantiated, x will be a part of it. So, x will get its memory through the instantiation of obj 1, but in case of static this is just a declaration, because when obj 1 will get instantiated the class variable x is not a part of this object.

So, I need to create the memory or bind the memory for this data member - static data member separately and this is what is known as a definition through which I create that. So, that definition says that it is done in the global scope, I say that this is the name of the variable MyClass :: x, class name :: variable name. The type is int. So, I say the type is int and I put the initialization which I have put here as 0. So, this is to define the static data member. So, if you write a program which has static data member, and if you do not do this, do not put this then the compiler will start crying saying that this particular static data member MyClass :: x has not been defined, and therefore will not get a memory.

So, this is something special that you will have to remember for the static data member that they will require a definition separately and they will need to be initialized when the program starts working.

So, when does this get executed, when does this x gets the value 0, it gets the value 0 before you start on the beginning of main. So, as we start executing the program, first all static data members of all classes will get constructed and initialized, and only then main will start working. Similarly when main ends only after that all the static data members of all classes will start getting destructed in the reverse order in which they were constructed. Of course, in this particular example, you do not see that construction, destruction process, because in this case the static data member is an object of the built in type, where we know there is no explicit constructor or destructor available.

(Refer Slide Time: 16:22)

```
#include <iostream>
using namespace std;
class PrintJobs { int nPages_; // # of pages in current job
public:
    static int nTrayPages_; // # of pages remaining in the tray
    static int nJobs_; // # of print jobs executing
    PrintJobs(int np) {
        nJobs_ = np;
        cout << "Printing " << np << " pages" << endl;
        nTrayPages_ = nTrayPages_ - np;
    }
    ~PrintJobs() { --nJobs_; }
};

int PrintJobs::nTrayPages_ = 800; // Definition and initialization == load paper
int PrintJobs::nJobs_ = 0; // Definition and initialization == no job to start with
int main() {
    cout << "Job = " << PrintJobs::nJobs_ << endl;
    cout << "Pages = " << PrintJobs::nTrayPages_ << endl;
    PrintJobs job1(10);
    cout << "Job = " << PrintJobs::nJobs_ << endl;
    cout << "Pages = " << PrintJobs::nTrayPages_ << endl;
    {
        PrintJobs job1(20);
        cout << "Job = " << PrintJobs::nJobs_ << endl;
        cout << "Pages = " << PrintJobs::nTrayPages_ << endl;
        PrintJobs::nTrayPages_ += 100; // Load 100 more pages
    }
    cout << "Job = " << PrintJobs::nJobs_ << endl;
    cout << "Pages = " << PrintJobs::nTrayPages_ << endl;
    return 0;
}

Output:
John = 0
Pages= 800
Printing 10 pages
Job = 1
Pages= 400
Printing 20 pages
Printing 20 pages
John = 3
Pages= 440
Job = 1
Pages= 840
```

So, with this let us move on, and look at a little bigger example, a more realistic example of using the static data member. So, here what we are trying to do we have we are creating a class print jobs, which has every printing job has a number of pages to print. And when I construct the print job object, then I know what are the number of pages that need to be printed in this job. Now what I want to track through this is I want to track that certainly I am assuming that all these print jobs actually fire the printing on a single printer. So, the tasks print jobs actually task go there. So, I want to track two information; as to, how many different print jobs are currently working, so I call that n jobs.

Now naturally how many jobs are currently ready for printing does not depend on the specific job, it depends on the total number of print job objects that have been constructed. So, it is a property which is at the class level, which is a static property and therefore, I call it a static int n jobs. Similarly, I conceive that as if my printer has a printing tray where a certain number of pages have been loaded, so I want to track how many pages are remaining in that tray. So, I create n tray pages data member which I also make static, because it is not specific to a particular job, but it is specific to the total print job class as to how many pages are remaining in the tray.

With that, now naturally here so these are the two static data members these are initialized constructed and initialize at this point and then I do a whole lot of tasks. I initially print the value of n jobs, which must be 0 here as the output, then I prints the number of pages that exist in the tray which is initialized with 500. So, the output should be 500, and then I construct a print job object. So, my number of jobs should get incremented by 1, and my number of pages remaining must get decremented by the number of pages that this job is supposed to print. So, it should get decremented by 10. So, after constructing this object, when I print the jobs and pages that are currently available, so I get jobs is now become 1 incremented from 0, and the number of pages remaining is 490 decremented by 10 from the 500 value.

Then I get into this particular scope, where I construct two more objects and then I see what is the value of the jobs, and the number of pages jobs will become 3, because two more print jobs are on, and the number of pages go down by 30 and 20 further. So, it becomes 30 less 460, 20 less from that, 440, so that is the remaining number of pages.

And then I increment the number of pages in the tray by 100, as if I am loading 100 pages, so my number of pages should become 440. And then I reach at this point and you can pretty well understand that since the scope is going out, this object job 1 and this object job 2 that were created within this scope will get destructed. So, if they get destructed then the number of jobs will come down from 3 to 1. So, when I print here I have one job, and when I print the number of pages remaining, I have loaded 100 more pages, so it becomes 540 in number, so that is how it behaves.

So, in this program using the static data members I can track some information which is not specific to every job, but it is specific to the whole collection of print jobs that I have that is the number of print jobs that are currently working; which is basically counting the number of objects of a class that is currently present in the system, and a global resource which is number of pages in the tray that I am manipulating.

(Refer Slide Time: 21:18)

The slide is titled "static Member Function". It lists several characteristics of static member functions:

- does not have this pointer – not associated with any object
- cannot access non-static data members
- cannot invoke non-static member functions
- can be accessed
 - with the class-name followed by the scope resolution operator (::)
 - as a member of any object of the class
- is needed to read / write static data members
 - Again, for encapsulation static data members should be private
 - get()-set() idiom is built for access (static member functions in public)
- may initialize static data members even before any object creation
- cannot co-exist with a non-static version of the same function
- cannot be declared as const

So, this is the typical use of the static data member. Now naturally as you can see that here the static data members are in the public visibility space, so as my application is changing it anyone can actually come and change these values by simply assigning a new value or changing on in terms of the incrementation or decrementation. So, what I should try to do next is to try to make them private, so that they cannot be changed directly.

Now certainly if I make them private then there comes a question of how do I manipulate it? I cannot change it anymore, because if I make it private then naturally I will not be able to change that value of the number of jobs or the number of pages in the tray. So, I need some function like, I have member functions, which can change data members, I need similar functions which can change the static data member values. And that brings us to the notion of static member functions.

Now, the critical thing about static member functions is just like functions. They are written with a static keyword in the front, but the difference is the key difference with other member function is they do not have a static member function, does not have at this pointer. Because, like the static data member is static member function is also not

associated with any object, it is just associated with the class since it is not associated with an object, it does not have the objects address of the this pointer.

So, the consequence of that is a next statement is since it does not have at this pointer, it cannot know the address of the data members of the object, because it is not referring to any objects. So, which means it cannot access non-static data members of the class. It cannot invoke the non-static member functions of the class because all of this both of this need this pointer. So, not at being at this pointer has a consequence that the static member function cannot access anyone of them.

So, how do you access a static member function, you like the static data member, you access the invoke the static member function using the class notation that is class name :: the scope resolution operator class name colon-colon, the member function name. Now, I have already started saying in terms of why we need these, because I wanted to read and write static data members.

Whereas, earlier in the example, we showed that the static data members are public which has encapsulation issues? So, if I encapsulate the static data members as private then I will be able to manipulate them, read and write them using the static member functions and static member functions since they are class specific they can access the class specific static data members and can change them together. So which means that static member functions along with static data members will again allow me to enforce a similar kind of encapsulation though in the class level not at the object level. And I can create similar kind of get set idiom with the static data member static member function and the static data members that we have.

It can few points to be noted that static data member does not have cannot co exist with a non-static member function. A static member function cannot co exist with a non-static member function of the same name that is you cannot have one member function of a class which is static, and by the same name have another which is non static, this is not allowed. And certainly the static data members cannot be const which is a consequence of not having this pointer, because we know that if a member function is const, and then basically the type of this pointer changes. This pointer becomes a pointer to a constant

object the static member function has no object to refer to therefore there is no sense of calling it a constant.

(Refer Slide Time: 25:21)

```
#include <iostream>
using namespace std;
class PrintJobs { int nPages; // # of pages in current job
    static int nTrayPages; // # of pages remaining in the tray
    static int nJobs; // # of print jobs executing
public: PrintJobs(int nP) : nPages_(nP), nJobs_(0) {
    cout << "Printing " << nP << " pages" << endl;
    nTrayPages_ = nTrayPages - nP;
}
~PrintJobs() { --nJobs; }
static int getJobs() { return nJobs; }
static int checkPages() { return nTrayPages; }
static void loadPages(int nP) { nTrayPages_ += nP; }
};

int PrintJobs::nTrayPages_ = 800; // Definition and initialization -- load paper
int PrintJobs::nJobs_ = 0; // Definition and initialization -- no job to start with
int main() {
    cout << "Jobs = " << PrintJobs::getJobs() << endl;
    cout << "Pages= " << PrintJobs::checkPages() << endl;
    PrintJobs job1(10);
    cout << "Jobs = " << PrintJobs::getJobs() << endl;
    cout << "Pages= " << PrintJobs::checkPages() << endl;
    PrintJobs job2(20);
    cout << "Jobs = " << PrintJobs::getJobs() << endl;
    cout << "Pages= " << PrintJobs::checkPages() << endl;
    PrintJobs::loadPages(100);
    cout << "Jobs = " << PrintJobs::getJobs() << endl;
    cout << "Pages= " << PrintJobs::checkPages() << endl;
    return 0;
}
```

NPTEL MOOCs Programming in C++

Partha Pratim Das

Output:

```
Jobs = 0
Pages= 800
Printing 10 pages
Jobs = 1
Pages= 400
Printing 20 pages
Jobs = 2
Pages= 400
Printing 20 pages
Jobs = 3
Pages= 500
```

With this, now let us go back and improve our print task example that we saw. So, what we have done here, we have moved the static members into the private part of the class declaration. So, now from main, you cannot directly change them it is not possible to change them directly, because they are private now. So, we have introduced a set of static member functions which can be used to read or write them, for example, getJobs reads, how many jobs are currently there; checkPages read, how many pages are remaining in the tray; loadPages, loads new pages on this and so on.

So, we rewrite this is the same application producing the same output, but instead of accessing these static data members directly, we now use the static member functions like when I want to know how many jobs are there I do `PrintJobs :: getJobs`. Again note the notation this is a class name :: the static member function name. When I want to check the number of pages I do `PrintJobs :: checkPages` and it will give me the number of pages remaining, I am sorry it will give me the number of pages remaining from here by invoking this function.

Certainly please note that none of these static member functions have this pointer. So, therefore they cannot access the non-static data like none of them can access the nPages data that may exist. They will only have to work with the static data that exist for the class.

(Refer Slide Time: 27:15)

The screenshot shows a presentation slide titled "Singleton Class". The slide content includes a bulleted list:

- A class is called a Singleton if it can have *only* one instance
- Many classes are singleton:
 - President of India
 - Prime Minister of India
 - Director of IIT Kharagpur
 - ...
- How to implement a Singleton Class?
- How to restrict that user can created *only* one instance?

The left sidebar lists navigation links: Module 16, Partha Pratim Das, Objectives & Outline, Static data member, Print Task, Member Function, Print Task, Singleton Class, and Summary.

You can go through this example at a later point you could line by line go through this example, and convince yourself that here we have the same functionality as the previous case, but we have been able to improve on the encapsulation. Now before we close, I would like to just quickly show you a very typical use of static members in terms of realizing, what we say is a singleton class.

A singleton class is a kind of design pattern which say that a class is called singleton, if you can have only one instance of that class at a time - only one instance at a time. Initially, it sounds little awkward absurd as to why do I need such classes. But if you think and look around, you will find number of classes have that kind of a behavior like president of India, there is only one president of India, there is only one prime minister of India, there is only one director of IIT, Kharagpur and so on, so several classes have only one instances.

Now if I want a class to be like that then how do I implement such class? Naturally it is easy to design a class which cannot have any instance. All that you need to is to make the constructor private. If you make the constructor private then nobody can call it, and therefore, you cannot have any instance of that class. But how do you make sure that you can construct only one object, but not more than one object, so that is something which can be done very easily using the static data member and the static member function.

(Refer Slide Time: 28:26)

```

Module 16
Partha Pratim Das
Objectives &
Outline
static data
member
Printer Test
private
Member
Function
Printer Test
Singleton
Class
Summary
Program 16.04: static Data & Member Function
Singleton Printer

#include <iostream>
using namespace std;

class Printer {
    /* THIS IS A SINGLETON PRINTER -- ONLY ONE INSTANCE */
    bool blackAndWhite_, bothSided_(b);
    cout << "Printer constructed" << endl; } // Private Printer cannot be constructed!
    static Printer *myPrinter_; // Instance of the Singleton Printer

public:
    Printer() { cout << "Printer destructed" << endl; }

    static const Printer* printer(bool bw = false, bool bs = false) { // Access the Printer
        if (!myPrinter_) {
            myPrinter_ = new Printer(bw, bs); }
        return myPrinter_; }

    void print(int np) const { cout << "Printing " << np << " pages" << endl; } // Output:
};

Printer *Printer::myPrinter_ = 0;
int main() {
    Printer::printer().print(10);
    Printer::printer().print(20);
    Printer::printer().print();
}

Printer::printer().Printer();
return 0;
}

```

NPTEL MOOCs Programming in C++ Partha Pratim Das 10

Printer constructed
Printing 10 pages
Printing 20 pages
Printer destructed

So, what you do is printer is one class. So, I am trying to show that how printer can be made singleton that is the situation is in the organization possibly I have only one printer. So, I have to make sure that more than one printer should never get instantiated. So, in the printer class, these are the non-static attributes these are just for completeness; do not worry about that, but here is the constructor which is made private. See this is in the private part. So, nobody directly can construct any object of the printer class type done.

The destructor of course, is kept in the public; because once the constructor is private you cannot construct it. So, destructor can still remain in the. Now what you do you create you introduce a static data member, which is a pointer to a printer, and also put that in private, so that nobody can directly. So, your idea is that this static data member which is Printer :: myprinter this one will point to the only printer object that I can have.

Now, the question of course, is how do I construct this only printer object. So, for that, in the public, you introduce a static member function. This member function is say called the printer. Now what it does, whenever you need a printer, whenever you need the printer object, you do not try to access the object directly, because you do not know where the object is.

Instead what you do, you invoke this particular static member function, what is static member function does, it initially checks, if this printer pointer is null or not null. What will happen at the very beginning here I have defined and initialized. So, at the very beginning this pointer will be null, there is no printer. If this is null you construct a printer and put the pointer here. So, you have constructed the printer and put it here.

Now mind you this particular static member function is able to construct the printer, because it is a member function. And therefore, it can access the private members. So, it invokes that gets a new printer, and returns you that printer. Subsequently, whenever you will call this particular static member function, you will find this to be non-zero. So, you will bypass this construction process, and you will always return the printer that you have constructed first.

In this process, only the first call will do a new on this printer object, and I will give you a printer object; from that point onwards, every time we will get back the same object. So, if you look into the use then this is you want to do the print job, so you say printer colon colon printer this part means this member function. So, this member function will then print, so it returns the printer mind you this is a return by reference. So, it returns the object printer, so on that object printer you invoke print which is this non-static member function and the printing will happen.

(Refer Slide Time: 32:31)

Module Summary

- Introduced `static` data member
- Introduced `static` member function
- Exposed to use of `static` members
- Singleton Class discussed

NPTEL MOOCs Programming in C++ Partha Pratim Das 13

Similarly, you can again do a print of 20. So, you can see this is the printer is got constructed, 10 pages are printed, 20 pages are printed and so on. And when you are done naturally you have created this object, so the responsibility is with you to destroy the object and therefore, you can call the destructor and destroy that object.

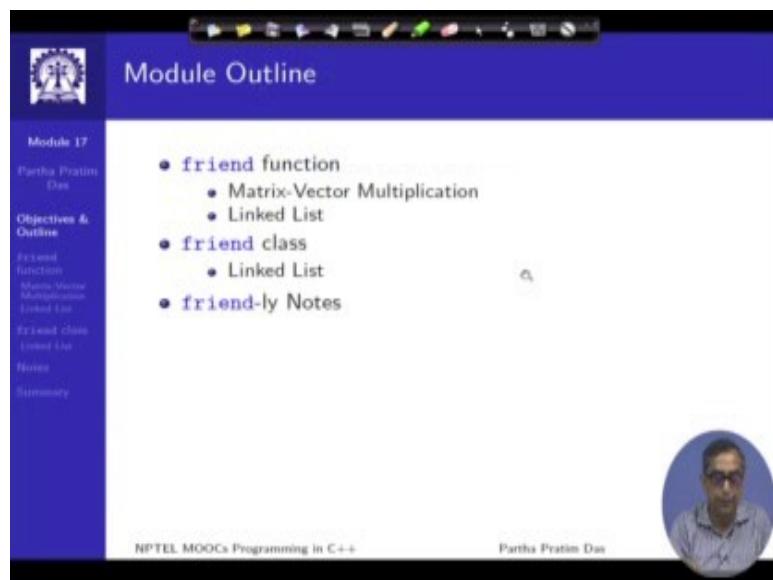
So, this is a simple way that singleton can be very safely implemented if you use static data members and static member functions. I have shown it with a printer class, but this can be done with any other class, which needs to be singleton. So, in summary, we have introduced the static data members and static member function, and we have shown that they can be used for various purposes of counting objects for maintaining any data at the class level and specifically for creating singleton objects.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 32
friend Function and friend Class

Welcome to module 17 of Programming in C++. In this module, we will talk about friend Function and friend Class trying to understand what do they mean and why are they useful in C++ design process.

(Refer Slide Time: 00:38)



Module Outline

- **friend** function
 - Matrix-Vector Multiplication
 - Linked List
- **friend** class
 - Linked List
- **friend-ly** Notes

NPTEL MOOCs Programming in C++ Partha Pratim Das

These will be the module outline we will take examples of Matrix Vector Multiplication and Linked List, and finally will end with some notes. As you know as usual the outline will be available on the left of your slide

(Refer Slide Time: 00:57)

Module 17

Partha Pratim Das

Objectives & Outcome

Friend function

Matrix Vector Multiplication
Linked List
Friend class
Friend func
Notes
Summary

NPTEL MOOCs Programming in C++

Partha Pratim Das

Ordinary function

```
#include<iostream>
using namespace std;

class MyClass { int data_; public: MyClass(int i) : data_(i) {} void display(const MyClass& a); }; void display(const MyClass& a) { cout << "data = " << a.data_; } // Error 1 int main(){ MyClass obj(10); display(obj); return 0; }
```

• display() is a non-member function
• Error 1: 'MyClass::data_': cannot access private member declared in class 'MyClass'

Friend function

```
#include<iostream>
using namespace std;

class MyClass { int data_; public: MyClass(int i) : data_(i) {} friend void display(const MyClass& a); }; void display(const MyClass& a) { cout << "data = " << a.data_; } // Okay int main(){ MyClass obj(10); display(obj); return 0; }
```

• display() is a non-member function
• friend to class MyClass
• Able to access data_, even though in class MyClass
• Output: data = 10

So, let us first introduce the basic notion of friend function. On the left you have the view of ordinary function. So the situation is like this, that I have a class myclass which as some private data, it has a constructor, and it has; I am sorry if just ignore this line. I have this function written outside of this class which is trying to do something with the class. What is it doing? It takes an object of the class by reference parameter call by reference, and then it tries to print the data element component of that objective.

Now, what do we know, this is private therefore, if I have a function outside I do not have the rights to access this object. So, just consider this is not there. This will give rise to an error and you will get an error of this kind cannot access the private member declared in myclass, because the reason you know very well by now is this is a global function and this is a private data so you cannot access this data directly.

Now, let us look into the right hand side. In the right hand side, we have exactly the same code except that we have introduced this in the class, much in the way we define the member function but with the difference that this is this inclusion is prefixed by a keyword friend. Now this does not make display a member function of the class, display is not a member function of the class, it continues to be a global function. A special type of a global function which is friend of myclass, but with this included when I say that

this global function display which takes an object a of myclass by reference and returns void, when I say that that is a friend of myclass then this does not remain an error this error disappears, this is now allowed.

So the concept is like this as if you know in our home also if somebody would step in we would normally allow them only to the drawing room, which is kind of the public place and we will not allow them to the inner rooms which is our private space. But if I have a friend stepping in then I will probably not keep him waiting in the drawing room rather I will take him to my inner rooms my bed room or kitchen and so on. So it is like a similar concept. So here, he has saying that this function the myclass is saying that the display function which is outside of this class which is not a member function is still a friend and therefore, the private data of this class will be exposed to this member. That is a basic idea of a friend function and therefore it will generate like this will give a compilation error but this will be a right, output will be produced.

(Refer Slide Time: 04:59)

The slide has a blue header bar with the title 'friend function'. On the left, there is a vertical sidebar with a logo at the top, followed by a navigation menu: 'Module 17', 'Partha Pratim Das', 'Objectives & Outline', 'Friend function', 'About Author', 'Module 18', 'Friend class', 'Friend fun.', 'Notes', and 'Summary'. The main content area contains two bullet points under the heading '● A friend function of a class':

- has access to the private and protected members of the class (breaks the encapsulation)
- must have its prototype included within the scope of the class prefixed with the keyword **friend**
- does not have its name qualified with the class scope
- is not called with an invoking object of the class
- can be declared **friend** in more than one classes

Below this, another bullet point under the heading '● A friend function can be a':

- global function
- a member function of a class
- a function template

At the bottom of the slide, there is footer text: 'NPTEL MOOCs Programming in C++', 'Partha Pratim Das', and a small number '5'.

With this let us see a formal definition of what a friend function is, “A friend function of a class has access to the private and protected members of the class.” That is it can break the encapsulation which have still not discussed what are protected members but, hold that for a while when we discussed that this will become clear but they are pretty much

like that private members only. So it must have a prototype included in the scope of the class and must be prefixed with friend, we have just in the example of the display function as to how do you write the signature of that function within the class with a friend in the front to say that this particular function is a friend of this class. Now this function does not belong to the class is not a member function therefore its name is not qualified by the class name as we do for normal non friend member functions.

It is not called with invoking an object because it is not a part of the class. A particular member function or a particular function can be friend of more than one class. Now, if you look at what all can be friend function; any global function could be a friend of a class, any member function of a different class could also be a friend of a class, or a function template could be a friend function. Now, function template again you do not know, but when you come to that you will understand how that works.

(Refer Slide Time: 06:37)

```

Module 17
Partha Pratim Das
Objectives & Outline
Friend Function
Matrix-Vector Multiplication
Index List
Friend class Index List
Home Summary

```

```

#include <iostream>
using namespace std;

class Matrix; // Forward declaration

class Vector {
    int n; // size of vector
public:
    Vector(int n) : n(n) {
        // Arbitrary initialization
        for (int i = 0; i < n; ++i)
            v[i] = i + 1;
    }
    void Clear() { // Set a zero vector
        for (int i = 0; i < n; ++i)
            v[i] = 0;
    }
    void Show() { //Show the vector
        for (int i = 0; i < n; ++i)
            cout << v[i] << " ";
        cout << endl << endl;
    }
    friend Vector Prod(Matrix *pM,
                       Vector *pV);
};

class Matrix {
    int m, n; // m rows, n columns
public:
    Matrix(int m, int n) : m(m), n(n) {
        // Arbitrary initialization
        for (int i = 0; i < m; ++i)
            for (int j = 0; j < n; ++j)
                a[i][j] = i + j;
    }
    void Show() { //Show the matrix
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j)
                cout << a[i][j] << " ";
            cout << endl;
        }
        cout << endl;
    }
    friend Vector Prod(Matrix *pM,
                       Vector *pV);
};

Vector Prod(Matrix *pM, Vector *pV) {
    Vector v(pM->n); v.Clear();
    for (int i = 0; i < pM->m; i++)
        for (int j = 0; j < pM->n; j++)
            v.v[i] += pM->a[i][j] * pV->a[j];
    return v;
}

// Vector Prod(Matrix*, Vector*); is a global function
// Vector Prod(Matrix*, Vector*); is friend of class Vector as well as class Matrix
// This function accesses the private data members of both these classes

```

Partha Pratim Das

With that, now let me introduce a more concrete problem. Here the problem is I have two classes, I have a vector class which keeps a linear vector of a size n, and have a matrix class which keeps a square matrix, not necessarily square matrix it is a keeps a matrix of m by n dimension. So, this is constructor for the vector just to keep things simple so that I do not have to enter the values of the vector here I have set as if the constructor itself

set certain values. Similarly, for the constructor of matrix as if when he construct the values are automatically initialized. I mean what values exist in that matrix or vector is not for our interest. What is of our interest is? Now given a vector object and a matrix object I want to define a function which will be able to compute the product of them. So I want a function like this.

A function prod which takes a matrix object here, I have taken it by a pointer takes a vector object multiplies them, of course following the rules of matrix vector multiplication and gives me the result which will of course in this case be a vector. What you see here is, this particular function prod and this is implementation I am not going through the implementation you can check it at a later point of time this is a typical matrix vector multiplication code. What I am interested to highlight is, a fact that this function here I have written it as a global function. This function needs to access the internal of the vector it will need to certainly access this array and the dimension. Similarly, this will also need to access the matrix and its dimensions to be able to actually compute their product.

Now, prod is not a member of either of these classes. In fact, making it a member does not solve the problem, because if I make prod a member of vector then it can access these easily, but it still cannot access them. If I make prod a member of matrix then I can access these but I cannot access this. So here I am providing a solution that I will make it a global function which is outside of each of these classes but, what I do I make prod a friend function in both these classes. If I have made a prod a friend of class vector then it necessarily means that prod will be able to access these private data members because it is a friend of a vector.

Similarly, I need to access the private members of matrix class, so I make prod a friend in the matrix class as well. Once I have done that then this code will compile because whenever I am making references to the private data members of vector or private data members of matrix both are accessible because prod is a friend of both these classes and this code will compile and if I write a main application with that.

(Refer Slide Time: 10:17)

Module 17

Partha Pratim Das

Objectives & Outcome

Friend Function

Matrix-Vector Multiplication

Friend class Prod

Notes

Summary

Program 17.02: Multiply a Matrix with a Vector

```
int main() {
    Matrix M(2, 3);
    Vector V(3);

    Vector PV = Prod(&M, &V);

    M.Show();
    V.Show();
    PV.Show();

    return 0;
}
```

Output:

0 1 2
1 2 3
1 2 3
8 16

- Vector Prod(Matrix*, Vector*) is a global function
- Vector Prod(Matrix*, Vector*) is friend of class Vector as well as class Matrix
- This function accesses the private data members of both these classes

NPTEL MOOCs Programming in C++

Partha Pratim Das

Where have I have constructed some matrix of dimension two three vector of dimension three as I am already mentioned that their values are getting filled in by default and this is what the matrix is filled up with, the so function shows that this is a vector is filled up with and then if I do the multiplication PV is the multiplication of the matrix and the vector and PV will turn out to be this output.

Here I could write this function, I would not to have been able to write this function unless I had the friend function feature. Because I can make prod a member of vector or I can make it a member of matrix but I certainly cannot make it a member of both, but to be able to compute the multiplication I need to access the members of both these classes. So, whenever we come across such situation where for completing functionality I need to access the internals or the private data members and methods of two or more independent classes I need the friend function feature to come in place.

(Refer Slide Time: 11:30)

```
#include <iostream>
using namespace std;

class Node; // Forward declaration

class List {
    Node *head; // Head of the list
    Node *tail; // Tail of the list
public:
    List(Node *h = 0);
    Node *head();
    Node *tail();
    void display();
    void append(Node *p);
};

class Node {
    int info; // Data of the node
    Node *next; // Ptr to next node
public:
    Node(int i): info(i), next(0) { }
    friend void List::display();
    friend void List::append(Node *p);
};

void List::display() {
    Node *ptr = head;
    while (ptr) {
        cout << ptr->info << " ";
        ptr = ptr->next;
    }
}

void List::append(Node *p) {
    if (!head) head = tail = p;
    else {
        tail->next = p;
        tail = tail->next;
    }
}

int main() {
    List l; // Init null list
    Node n1(1), n2(2), n3(3); // Few nodes
    l.append(&n1);
    l.append(&n2);
    l.append(&n3);
    l.display(); // Show list
    return 0;
}
```

This is just another example here; the example is I am trying to build a list. So how do I build a list? I have a node class. A node class defines basic node information which has a data part and a link part to link to the next node in the list. And I have a list class, which actually holds this whole list so it is like this. The list class has as you can see two pointers, the header pointer and the tail pointer so in general it will look something like this, I will have a list like this, say this is 2 this is 3 and let say the list has ended here. So, my head will point to the beginning of the list and tail will point to the end of the list. So I want to actually make such a structure and then I would like to use the list as I want.

Suppose, if I want to implement these kind of functionality in the list that is display, that is go over the list and output all the elements that you have or you append to the list. If I want to write say list : : display, this is a member function of list. So if I want to implement this then I need the internal information of the node, because unless I can access the info of the node I cannot print that value. Unless I can access the next field of the node I cannot go to the next node on the list so I need to access them.

But display basically is a member function of the list class, so it does not allow me to access the private members of the node class and that is where I make use of the friend. What I do? In the node class I write the signature of the display function. Note the

display is a member function so the name of the display function is actually `list :: node`, so I am writing here as I am sorry, the name of the member function is `display` and the full name is `list :: display`. So, I am writing the whole signature here and I prefix with the keyword `friend`.

Both the `display` and `append` which are member functions of `list` are made into friend of the `node` class, so that means that when I try to implement `display` or I try to implement `append` they will be able to access the private data members of the `node` class and I will be able to get a very nice `list` implementation in place, and if there is an application simply which creates a null `list` creates some `node` objects and then it appends the `node` object one after the other and it will finally print the list. If you do it you will get a printing of 1 2 3 because we have created 3 nodes with value 1 two and 3, that is not very important.

What I wanted to show that using the `friend` function you can actually very nicely link up to such classes which are somewhat related but, completely independent one which gives the primitive `node` that you want to keep the value in and the other that actually manages the `list` structure, manages the up and delete, `display` kind of functionality in them. And we are cutting across their encapsulation boundary we are creating a whole function in the encapsulation of `node` by making some of the member functions of the `list` class as friend of the `node` class.

Compared to the earlier example of matrix vector multiplication which showed how a global function can be made a friend of two classes, here I am necessarily shown that a member function, actually two member functions of one class has been made friend of another class. The matrix vector multiplication can also be solved in this style that I could have made `prod` a member function of `vector` and made that `vector :: prod` a friend of `matrix` or vice versa that is make `prod` a member function of `matrix` and then make `matrix :: prod` a friend of `vector`.

This is typically how the `friend` function feature can be used for better advantage.

(Refer Slide Time: 16:18)

The screenshot shows a presentation slide titled "friend class". The slide content includes:

- A **friend** class of a class
 - has access to the private and protected members of the class (breaks the encapsulation)
 - does not have its name qualified with the class scope (not a nested class)
 - can be declared **friend** in more than one classes
- A **friend** class can be a
 - class
 - class template

Below the text is a hand-drawn diagram consisting of two simple shapes: a rectangle and a circle, connected by a line.

The slide footer includes the text "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

There is certain bit of extension in this, that extension is driven from the fact that if you just think about the list and node example, then we needed both the member functions of list append as well as display to be friend of node. Now, if the list classes are delete function it will also need to be friend of node and so on. There is a shortcut that you can do and that is in terms of a friend class. So if I have two classes then we can make one class as a friend of the other by simply including that class name and prefixing with friend. If a class is a friend of the other then it has access to the private and protected members of the class. For a friend function that particular function has access, for a friend class all methods of that class have access to the private data members and member functions of the friend class irrespective of which member function I am talking of.

Naturally, since this is a class as a whole this class mean is not qualified by the name of the class to which it is the friend, because it is not a nested class it is just that is two independent classes, I have one class here, I have one class here, I have c 1 here, I have c 2 here. I am just saying that c 2 is a friend of c 1 therefore, their name will remain to be their isolated name and certainly a class can be declared as friend in more than one other classes. What can be a friend class of course, a class can be a friend class and a class template can be a friend class. Class template we have not done yet when we complete

talk about template this point will become clear as to how class templates also become friend classes.

(Refer Slide Time: 18:23)

```

Module 17
Partha Pratim Das
Objectives & Outline
Friend Function
Matrix Vector
Linked List
Friend class
Linked List
Notes
Summary

#include <iostream>
using namespace std;

class Node; // Forward declaration

class List {
    Node *head; // Head of the list
    Node *tail; // Tail of the list
public:
    List(Node *h = 0) : head(h), tail(h) {}
    void head();
    void tail();
    void display();
    void append(Node *p);
};

class Node {
    int info; // Data of the node
    Node *next; // Ptr to next node
public:
    Node(int i, Node *n) : info(i), next(n) {}
    //friend void List::display();
    //friend void List::append(Node *);
    friend class List;
};

void List::display() {
    Node *ptr = head;
    while (ptr) {
        cout << ptr->info << " ";
        ptr = ptr->next;
    }
}

void List::append(Node *p) {
    if (!head) head = tail = p;
    else {
        tail->next = p;
        tail = tail->next;
    }
}

int main() {
    List l; // Init null list
    Node n1(1), n2(2), n3(3); // Few nodes
    l.append(&n1);
    l.append(&n2);
    l.append(&n3);

    l.display(); // Show list
    return 0;
}

```

* List class is now a friend of Node class. Hence it has full visibility into the internals of
* When multiple member functions need to be friends, it is better to use friend class

NPTEL MOOCs Programming in C++ Partha Pratim Das

Now, let us rework the link list example by using the friend class concept. So what I am done in the same earlier example I have just commented out the two lines which we are making the member function display and the member function append of the list class as friends of the node class, instead I have made the whole list class as a friend. This is the way to write it the class and the class name is the way you refer and then you put a friend keyword in front to say that this node class considers that the whole of the list class is it is friend. It not only display and append (Refer Time: 19:14) wherever whatever other members like delete, like search, whatever member that we might add to the list class all of them will be accepted as a friend of the node class, rest of the implementation is same. Rest of the implementation and this application is no different from earlier.

The only difference is, in these three lines where we instead of using friend function we are using a friend class. So, particularly if two classes have too many member functions to be made friend of then we should consider whether instead of individually making member functions as friends, whether you should actually make the whole classes friend

and that will make often make your life easier because you do not have to specifically keeps on listing all the different member functions as friend.

(Refer Slide Time: 20:11)

The slide has a blue header with the title 'friend-ly Notes'. On the left, there's a sidebar with a logo, the title 'Module 17', and a list of topics: Objectives & Outline, Friend Function, Matrix-Vector Multiplication, United List, Friend class, United List, Notes, and Summary. The main content area contains two bullet points:

- **friend-ship** is neither commutative nor transitive
 - A is a friend of B does not imply that B is a friend of A
 - A is a friend of B and B is a friend of C does not imply that A is a friend of C
- **Visibility and Encapsulation**
 - **public**: a declaration that is accessible to all
 - **protected**: a declaration that is accessible only to the class itself and its subclasses
 - **private**: a declaration that is accessible only to the class itself
 - **friend**: a declaration that is accessible only to **friend**'s of a class.
friend's tend to break data hiding and should be used judiciously.
Like:
 - A function needs to access the internals of two (or more) independent classes (Matrix-Vector Multiplication)
 - A class is built on top of another (List-Node Access)
 - Certain situations of operator overloading (like stream operators)

At the bottom right, there's a circular profile picture of Partha Pratim Das. The footer includes the text 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

So, now you have seen how this friend stuff works. Now let me end with a couple of friendly nodes first point is to be noted, is friendship that is when we say is a one class is a friend of other this friendship is kind of a binary relationship between two classes and this binary relationship is neither commutative is nor transitive. It is not commutative which means that if A is a friend of B then that does not mean that B is a friend of A. If I have to make B is a friend of A then within the class scope of A I will have to put friend class B this statement, but by the mere fact that A is a friend of B does not imply that B is a friend of A, so it is not a commutative relation. Similarly, if I have A as a friend of B, B is a friend of C from within these classes that does not imply that A is a friend of C, so the transitivity does not work. Friendship is just binary, is just work between two classes and no further inferencing is possible.

Now having said this, please note that this friend feature of friend function and friend class actually changes the encapsulation and the visibility structure of the language. So far we have had three kinds of visibility of which two we have already discussed public and private, we will soon discuss protected visibility which applies to inheritance. But

friend is a fourth kind of visibility that exist in C++ where you can specifically one class can make some other classes, some other member function a friend and thereby giving it a visibility which is restricted, but is kind of puncturing the whole of encapsulation here. So, it should be what you make friend should be done very judiciously because if you arbitrarily make other classes and other functions, global functions or member functions as friends then all your advantages of encapsulating data and using them accessing them through a proper choice of member functions will get lost.

So, herein we have tried to put a few situations which are common where the friend using friend really helps. First is the matrix vector kind of situation where you have two independent classes, but you have a functionality where the data members, private members of both these classes participate as in case of multiplying a matrix with a vector. Or you have a situation where one class is being built on top of another class as a design component like in a list, list comprise of different node class objects so the list functionality certainly gets far easier to implement if you use node, if you allow node to announce list as a friend of it so that it can look through the whole of the internal of the list. This built up is another situation.

The third which we have not yet actually discussed but I would just put a pointer that, when we tried to overload operators their certain operators which are very difficult to overload with proper syntax and semantics if we did not have friend function kind of functionality available in C++. Example we will show in terms of output input streaming operators when you overload them for specific user defined classes, but in general you should be very cautious, restrictive and conservative in terms of when you use friend function and friend class and you should really make sure that you have one of these different situations and may be some of the very related situations to this really occurring, but otherwise if you just use the friend feature as a function or as a friend of class just to shortcut the design then you are actually breaking down the encapsulation which will go against the basic object oriented framework that we are building up so carefully through the definition of access specifiers and the creation of the objects and so on.

So, friend is a powerful feature and like any powerful feature, like any powerful weapon it must be used with a lot of care and in discretion.

(Refer Slide Time: 25:09)

The screenshot shows a presentation slide titled "Module Summary" for "Module 17". The slide content is as follows:

- Introduced the notion of **friend** function
- Introduced the notion of **friend** class
- Studied the use of **friend** function and **friend** class with examples
- **friend** introduces visibility hole by breaking encapsulation
 - should be used with care

On the right side of the slide, there is a circular video frame showing a person with glasses and a light-colored shirt, likely the lecturer, Partha Pratim Das. The video frame has a blue border. At the bottom of the slide, there is a footer with the text "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

To summarize in this module we have introduced the notion of friend function and the notion of friend class and we have studied the friend function and friend class with examples of matrix multiplication, and risk manipulation, and we have noted specifically that friend is a different kind of visibility and somewhat dangerous, somewhat risky to use because it can arbitrarily break the encapsulation and therefore the use of friend should be done with very judicious choice of design with proper design justification of why this punctures is required.

As you go ahead and start doing lot of designs and implementation we will find that kind of almost always that you need to use a friend it will be one of the three situations that I have discussed here, and if you find that you are requiring a friend function or a friend class to be used in a situation which is not like the three that we have discussed then you should be very cautious and careful and really convince yourself that this is a situation which needs the friend to be used.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 33
Overloading Operator for User - Defined Types: Part 1

Welcome to Module 18 of Programming in C++. We have earlier talked about operator overloading. We had discussed that operators in C++, majority of them can be associated with corresponding operator functions that can be overloaded defined by the user. In this module and the next one, we would take a deep look into operator overloading for user-defined types it is a connected discussion. So, this will be the part-1 of this discussion.

(Refer Slide Time: 01:09)

The screenshot shows a presentation slide titled "Module Objectives". The slide content is as follows:

Module Objectives

- Understand how to overload operators for a user-defined type (class)
- Understand the aspects of overloading by global function and member function

The left sidebar contains a navigation menu with the following items:

- Module 18
- Partha Pratim Das
- Objectives & Outline
- Motivation
- Operator Function
- Using global function
 - private data members
 - private data members
 - Using member function
 - operators
 - operators
 - User Overload
- Summary

At the bottom of the slide, there is footer text: "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

We will try to understand how to overload operators for user-defined types. And we would explore the aspects of overloading using global functions and member functions in this module.

(Refer Slide Time: 01:27)

The screenshot shows a presentation slide titled "Module Outline". On the left, there is a vertical navigation menu with the following items: Module 18, Partha Pratim Das, Objectives & Outline, Motivation, Operator Function, Using global function, Using member function, Summary. Below these, there are sections for public data members and private data members, each with three sub-points: operator+, operator=, and Unary operators. At the bottom of the slide, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das". A small circular video player window in the bottom right corner shows a man speaking.

The outline as usual would be available on the left of your screen.

(Refer Slide Time: 01:36)

The screenshot shows a presentation slide titled "Motivation". On the left, there is a vertical navigation menu with the following items: Module 18, Partha Pratim Das, Objectives & Outline, Motivation, Operator Function, Using global function, Using member function, Summary. Below these, there are sections for public data members and private data members, each with three sub-points: operator+, operator=, and Unary operators. The main content area lists several bullet points about operator overloading:

- We have seen how **overloading operator+** a C-string wrapped in struct allows us a compact notation for concatenation of two strings (Module 09)
- We have seen how **overloading operator=** can define the deep / shallow copy for a UDT and / or help with user-defined copy semantics (Module 14)
- In general, operator overloading helps us build complete algebra for UDT's much in the same line as is available for built-in types:
 - **Complex type:** Add (+), Subtract (-), Multiply (*), Divide (/), Conjugate (!), Compare (==, !=, ...), etc.
 - **Fraction type:** Add (+), Subtract (-), Multiply (*), Divide (/), Normalize (unary *), Compare (==, !=, ...), etc.
 - **Matrix type:** Add (+), Subtract (-), Multiply (*), Divide (/), Invert (!), Compare (==), etc.
 - **Set type:** Union (+), Difference (-), Intersection (*), Subset (<, <=), Superset (>, >=), Compare (==, !=), etc.
 - **Direct IO:** read (<<) and write (>>) for all types
- Advanced examples include:
 - **Smart Pointers:** De-reference (unary *), Indirection (= >=), Compare (==, !=), etc.
 - **Function Objects or Functors:** Invocation (())

At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das". A small circular video player window in the bottom right corner shows a man speaking.

Now, before we actually start the discussions on operator overloading, let me quickly summarize recap as to why we want operators to be overloaded. I would like to refer to two earlier modules; module 9, where we discussed about overloading operator + for

structured types, enumerated types to show that the operator + can be overloaded for a string structure to mean concatenation of strings. Similarly, we saw specific overloading for enum types and so on. Later in module 14, we talked about that operator, overloading for assignment operator is critical to define proper semantics for copy specifically with reference to deep and shallow copy.

In general, as we know operators and functions of the same purpose, but the difference is the operator has a very compact notation, which typically is infix. So, it is possible to write a combination of quite varied complex operations in terms of a compact expression; if I can define proper operator for the proper functionality that we want to do. So, these are by default available for built in types like int, like double some operators are available for pointer type as well and so on.

So, if I can overload the operators for a type that we are going to define; that is a user-defined type; much in the way that the operators are defined for the built in types. Then for user-defined type, also we can write expressions, we can write actually build a complete algebra based on the particular user-defined type or UDT that we are building.

So, here I have just mentioned and tried to give a very aboard outline, in terms of what can be done. For example, the C++ does not have a complex type. It has double type, but it does not have a complex type. But using operator overloading, we can define like the operator ‘+’ we can say is the add of complex numbers, this is subtraction of complex number, multiplication of complex number, division of complex number. All those can be defined in terms of the corresponding operators and something, which is specific for complex type like finding the complex conjugate. We can use the ‘!’ operator; overload the exclamation operator to mean that the sense being that ‘!’ operation is kind of a negation. So, complex conjugate is kind of a negation of that operator and so on.

Similarly, we can have fraction types, we can have matrix types where we can have a operators for all matrix a algebra including inversion of matrices, we can have set types with union, difference, superset, subset relations and so on. So, all of these and variety of different types that you may want to need to define create for as UDT you can build complete types using operator overloading. And particularly, in terms of the IO’s, we can

do very compact IO in terms of the operator output and operator input streaming. Those operators could also be overloaded for the UDT's to give very compact IO structures.

And I would just like to mention, but it is somewhat at the at an advance level. That some of the very nice designs in C++ of a smart pointer, of functors and so on are base significantly on operator overloading.

(Refer Slide Time: 05:45)

Operator Functions in C++: RECAP (Module 9)

Operator Expression	Operator Function
a + b	operator+(a, b)
a = b	operator=(a, b)
c = a + b	operator=(c, operator+(a, b))

- Introduces a new keyword: `operator`
- Every operator is associated with an operator function that defines its behavior
- Operator functions are implicit for predefined operators of built-in types and cannot be redefined
- An operator function may have a signature as:

```
MyType a, b; // An enum or struct  
// Operator function  
MyType operator+(const MyType&, const MyType&);  
a + b // Calls operator+(a, b)
```
- C++ allows users to define an operator function and overload it

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, overloading an operator is a critical requirement for building of a good user-defined type. So, let us just go forward this is a quick recap from module 9, we had observed that for every operator, there is an operator function, which we can define. And that operator function has certain a signature like the operator '+'. In this case, we have already seen this; I will not waste a lot of time here.

(Refer Slide Time: 06:15)

The slide has a blue header with the title 'Non-Member Operator Function'. On the left, there's a sidebar with a logo and navigation links for 'Module 18', 'Partha Pratim Das', 'Objectives & Outline', 'Motivation', 'Operator Function', 'Using global function', 'operator', 'operator=', 'Unary Operators', 'Summary'. The main content area contains bullet points and code snippets:

- A non-member operator function may be a
 - Global Function
 - friend Function
- **Binary Operator:**
MyType a, b; // An enum, struct or class
MyType operator+(const MyType&, const MyType&); // Global
friend MyType operator+(const MyType&, const MyType&); // Friend
- **Unary Operator:**
MyType operator++(const MyType&); // Global
friend MyType operator++(const MyType&); // Friend
- **Note:** The parameters may not be constant and may be passed by value.
The return may also be by reference and may be constant
- **Examples:**

Operator Expression	Operator Function
a + b	operator+(a, b)
a = b	operator=(a, b)
++a	operator++(a)
a++	operator++(a, int) Special Case
c = a + b	operator=(c, operator+(a, b))

At the bottom right is a circular portrait of Partha Pratim Das.

Just recap, in case it has become hazy in your mind. Now this operator function could be a non-member function. Like it could be a global function and we know that there are friend functions also. So, if a operator function is a global function or a friend function then it could look something like, this is a global operator function for adding two objects of my type whatever that my type is. Similarly, for a - if it is friend, then it will look something like this only difference being the friend keyword coming in.

Similarly, we can have global or friend functions for unary operators these are for binary operators. So, you can have it for unary operators. We can have specific operators for a prefix operator and postfix operator and so on.

(Refer Slide Time: 07:10)

The slide is titled "Member Operator Function". It includes a sidebar with navigation links for Module 1B, Partha Pratim Das, Objectives & Outcome, Motivation, Operator Function, Using global function, pointer data, pointer to pointer data, Using member function, operator+, operator++, Unary Operators, and Summary. The main content area has a blue header with bullet points: "● Binary Operator:", "MyType a, b; // MYType is a Class", "MyType operator+(const MyType&); // Operator function"; "● The left operand is the invoking object – right is taken as a parameter"; "● Unary Operator:"; "MyType operator-(); // Operator function for Unary minus", "MyType operator++(); // For Pre-Incrementer", "MyType operator++(int); // For post-Incrementer"; "● The only operand is the invoking object"; "● Note: The parameters may not be constant and may be passed by value. The return may also be by reference and may be constant"; "● Examples:"; and a table comparing Operator Expression and Operator Function. The table shows examples like a + b (operator+(b)), a = b (operator=(b)), ++a (operator++()), a++ (operator++(int) // Special Case), and c = a + b (operator =(a.operator+(b))). A small video thumbnail of Partha Pratim Das is in the bottom right corner.

In addition, we could also have operator functions, which are basically member functions. So, if it is a member function then the operator function will look something like this. A one major point to note here that '+' is a binary operator, so it needs two operands. But here you are you will see only one parameter for operator +, because since this a operator function is a member function the first operand or the left hand side operand the left operand of the operator is a object on which you are invoking the operator function. So, as you know that if I have a member function then in here I have the invisible this pointer for every member function.

So, this invisible this pointer actually means that my first operand is star this or the current object that I have, that always will be the left operands and the right operand is what is passed through here. So, in case of member function this is a difference from the global or friend function.

The unary operators can also be defined, in a way naturally unary operators will not have an operand, because they have only one and that is the object on which you are invoking it. With the exception that if it is a post fixed operator, then we will have an operand of type int that actually will not have an operand? But the signature has an additional type int this means that it is a post fix operators. So, that is used to distinguish between the

prefix operators which is this one and the post fix operator, because both of these by name are ‘++’. So, this additional type in the signature list designates that which one is the prefix operator, which one is the post fix operator. So, these are the different options through which we can actually define the operator functions and overload them.

(Refer Slide Time: 09:15)

The slide has a blue header with the title 'Operator Overloading – Summary of Rules: RECAP (Module 9)'. On the left, there is a vertical sidebar with a navigation menu:

- Module 18
- Partha Pratim Das
- Objectives & Outline
- Motivation
- Operator Function
- Using global function
- Using member function
- Summary

The main content area contains a bulleted list of rules for operator overloading:

- No new operator such as **, <>, or &| can be defined for overloading
- Intrinsic properties of the overloaded operator cannot be changed
 - Preserves arity
 - Preserves precedence
 - Preserves associativity
- These operators can be overloaded:
[], +, -, *, /, %, &, |, ~, !=, ==, *=, /=, %=, &=, |=,
<<, >>, >>=, <<=, !=, <, >, <<, >>, &&, ||, ++, --, ->*, ->, (), []
- The operators :: (scope resolution), . (member access), * (member access through pointer to member), sizeof, and ?: (ternary conditional) cannot be overloaded
- The overloads of operators &&, ||, and , (comma) lose their special properties: short-circuit evaluation and sequencing
- The overload of operator-> must either return a raw pointer or return an object (by reference or by value), for which operator-> is in turn overloaded
- For a member operator function, invoking object is passed implicitly as the left operand but the right operand is passed explicitly
- For a non-member operator function (Global/friend) operands are always passed explicitly

So, I would again quickly refer you to module 9. Where, we saw the summary of rules of what we can do for operators. We cannot change the basic properties of the operators and there is a fixed list of operators which only can be overloaded there are some operators which cannot be overloaded, some that should not be overloaded, we discuss this in depth on module 9, so I will skip that. Now what is additional these two points which have highlighted in blue line is for a member function invoking object is passed implicitly which I have just explained and that turns out to be the left operand, so only the right operand is explicitly mentioned. And in case of global function or friend function certainly both the operands are explicitly mentioned.

(Refer Slide Time: 10:10)

The slide title is "Program 18.01: Using Global Function – Unsafe (public Data members)".

Complex Addition Code:

```
#include <iostream>
using namespace std;
struct complex { // public data member
    double re;
    double im;
};

complex operator+ (complex &a, complex &b) {
    complex r;
    r.re = a.re + b.re;
    r.im = a.im + b.im;
    return r;
}

int main() {
    complex d1 , d2 , d;
    d1.re = 10.5; d1.im = 12.25;
    d2.re = 20.5; d2.im = 30.25;
    d = d1 + d2;
    cout << "Real: " << d.re;
    cout << "Imaginary: " << d.im;
    return 0;
}
```

String Concatenation Code:

```
#include <iostream>
#include <cstring>
using namespace std;
typedef struct _String { char *str; } String;
String operator+(const String &s1,
                   const String &s2) {
    String s;
    s.str = (char *) malloc(strlen(s1.str) +
                           strlen(s2.str) + 1);
    strcpy(s.str, s1.str);
    strcat(s.str, s2.str);
    return s;
}

int main() {
    String fName, lName, name;
    fName.str = strdup("Partha ");
    lName.str = strdup("Das");
    name = fName + lName; // Overload operator +
    cout << "First Name: " << fName.str << endl;
    cout << "Last Name: " << lName.str << endl;
    cout << "Full Name: " << name.str << endl;
    return 0;
}
```

Output: Real: 31, Imaginary: 42.5
operator+ is overloaded to perform addition of two complex numbers which are of struct complex type

Output: First Name: Partha, Last Name: Das
Full name: Partha Das
operator+ is overloaded to perform concat of first name and last to form full name. The data type is struct String

So, this is a basic of operator overloading. So, with this now let us a look back into some of the code that we had seen earlier is to try to overload an operator using a global function. So, I have a simple a complex class which I have written in term of a structure. So, which means that both it is members are publicly available and we have written operator + addition operator for this complex structure, which is basically component wise addition of the real and imaginary parts. And if we use that in this expression d1 is being a complex, d2 being another complex we can write d1+d2. To mean addition of two complex numbers, which will actually this operator we will actually invoke this function the operations will happened and that result will get assigned to d. So, this is fine.

I am sure you will not have a much difficulty in understanding that. On the right hand side you have another example, where we are using a structure to wrap a C string here and based on that structure type, the string structure type, we have defined and overloading for the operator ‘+’ again for the string. So, that given two strings here we can write add expression for the first name and the last name which actually goes to this function and performance a concatenation. So, depending on the kind of a type we have the same operator + in some case is may being addition of complex numbers, in some case it means at the concatenation of strings and so on. This is quite straightforward to do

and for this, we have used the global functions here.

(Refer Slide Time: 12:06)

Module 18
Partha Pratim Das
Objectives & Outline
Motivation
Operator Function
Using global function
public data members
private data members
Using member function
operator<</operator>>
Summary

```
#include <iostream>
using namespace std;
class Complex { // Private data members
    double re, im;
public:
    Complex(double a=0.0, double b=0.0):
        re(a), im(b) {}
    Complex() {}
    void display();
    double real() const { return re; }
    double img() const { return im; }
    double set_real(double r) { re = r; }
    double set_img(double i) { im = i; }
};
void Complex::display() {
    cout << re;
    cout << " + " << im << endl;
}

Complex operator+(Complex &t1, Complex &t2) {
    Complex sum;
    sum.set_real(t1.real() + t2.real());
    sum.set_img(t1.img() + t2.img());
    return sum;
}

int main() {
    Complex c1(4.5, 25.25), c2(0.3, 10.25), c3;
    cout << "1st complex No: ";
    c1.display();
    cout << "2nd complex No: ";
    c2.display();
    c3 = c1 + c2;
    cout << "Sum = ";
    c3.display();
    return 0;
}
```

Output:

```
1st complex No: 4.5 + 25.25
2nd complex No: 0.3 + 10.25
Sum = 4.8 + 35.5
```

- Accessing private data members inside operator functions is clumsy
- Critical data members need to be exposed (get/set) violating encapsulation
- Solution: Member operator function or friend operator function

NPTEL MOOCs Programming in C++
Partha Pratim Das



Now naturally, if you use a global function then we know that is not an advisable thing a in object oriented programming system, because the global functions, but to work they naturally needed that the data members the particularly the real and imaginary part of the complex number had to be made public and are we have repeated taken this principle that the data member should actually be private. So, we should make them private. Only the different operations that the class supports, only those should be public.

Now if we do that then naturally a certainly with this we cannot write the operator + overwrite the operator + as a global function directly, because this global function will not able to access the private data members. So, what will have to do for making this work, we have to add a bunch of get set functions for the components. Where I can read the real component, read the imaginary component, write the real component write the imaginary component.

Therefore, then using this get set methods we can write a global operator overloading for the operator +. This works this will work fine this will work as it is and we have been able to make the data members private, but this tell does not pretty much give a better

solution. Because, in order to implement the global function we had to provide all these get set function. Then the more of the get set functions you provide your actually exposing your internal data to the external world.

Because, well people cannot directly access the data members and, but since the operator + can access set and get a real and imaginary parts any other a external function any external class can also do that. So, in a way this solution is kind of breaking the encapsulation that we have. So, certainly we know with global functions we cannot preserve encapsulation.

(Refer Slide Time: 14:17)

Module 18

Partha Pratim Das

Objectives & Outline
Motivation
Operator Function
Using global function
private data members
private data members
Using member function
operator+ operator<
Using Operator Overloading Summary

Program 18.03: Using Member Function

```
#include <iostream>
using namespace std;
class Complex { // Private data members
    double re, im;
public:
    Complex(double a=0.0, double b=0.0):
        re(a), im(b) {}
    ~Complex() {}
    void display();
    Complex operator+(const Complex &c2) {
        Complex r1;
        r1.re = re + c2.re;
        r1.im = im + c2.im;
        return r1;
    }
};

void Complex::display(){
    cout << re;
    cout << "j" << im << endl;
}

int main() {
    Complex c1(4.5, 25.25), c2(0.3, 10.25), c3;
    c1.display();
    cout << "2nd complex No:";
    c2.display();
    c3 = c1 + c2;
    cout << "Sum = ";
    c3.display();
    return 0;
}
```

Output:

```
1st complex No: 4.5 +j 25.25
2nd complex No: 0.3 +j 10.25
Sum = 12.8 +j 35.50
```

■ Performing $c1 + c2$ is equivalent to $c1.operator+(c2)$
 ■ $c1$ invokes the $operator+$ function and $c2$ is passed as an argument
 ■ Similarly we can implement all binary operators ($%, *, \text{ etc.}$)
 ■ Note: No need of two arguments in overloading

NPTEL MOOCs Programming in C++

Partha Pratim Das

The next option we will look into which is basically using member functions and for doing the same task. Let us now consider that the operator now is overloaded as a member function. So, you can see that it has moved inside the class. My data members are still private and since it is moved inside the class, it has only one parameter, which is the right hand side parameter. And the left hand side parameter would be the object itself. So, when I want to do add I, add re with c.re. Where, c is the right hand operand and re as you know refers to the real component of the current object on which this method has been invoked, which has to be the left hand side operand.

So, as in a here if you have written $c1 + c2$ as we write $c1+c2$ this implies that the notation would be that this is equivalent to $c1$ ‘operator +’ $c2$. So, which means that the $c2$ here is a right hand side operand which will become c and the $c1$ here is the current object which is star this. That is when we are referring to re we are actually referring to the real component of $c1$, when I referring to $c.re$ we are referring to the real component of $c2$ and that is how this computation will go through.

Certainly this again gives you the same a solution this gives you a same answer it has advantage of a making a protecting the encapsulation that we want to preserve. So, we can a very easily work with this and have good operator overloading using the member functions.

(Refer Slide Time: 16:05)

```

Module 18
Partha Pratim Das
Objectives & Outline
Memory
Operator Function
Using global function
private data members
private data members
Using member function
operator<
operator>
Summary

Program 14.14: Overloading operator=
RECAP (Module 14)

#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class String { public: char *str_; size_t len_};

String(char *s) : str_(strdup(s)), len_(strlen(str_)) {} // ctor
String(const String& s) : str_(strdup(s.str_)), len_(s.len_) {} // copy ctor
~String() { free(str_); } // dtor
String& operator=(const String& s) {
    if (this != &s) {
        free(str_);
        str_ = strdup(s.str_);
        len_ = s.len_;
    }
    return *this;
}
void print() { cout << "(" << str_ << ", " << len_ << ")" << endl; }

int main() { String s1 = "Football"; s2 = "Cricket";
    s1.print(); s2.print();
    s1 = s1; s1.print();
    return 0;
}

(Football: 8)
(Cricket: 7)
(Football: 8)

    • Check for self-copy (this != &s)
    • In case of self-copy, do nothing
NPTEL MOOCs Programming in C++
Partha Pratim Das

```

Now in the same way the as you have seen shown for the binary addition operator several, other operators can be overloaded. For example, if you recall in module 14 while we discussed about various opportunities in options of copying, we showed that operator assignment copy assignment operator. We had been talking about can be overloaded and this is we showed that is a safe way of doing this whole example is actually again repeated from module 14 for your reference.

(Refer Slide Time: 16:45)

Notes on Overloading operator=
RECAP (Module 14)

- Overloaded operator= may choose between Deep and Shallow Copy for Pointer Members
 - Deep copy allocates new space for the contents and copies the pointed data
 - Shallow copy merely copies the pointer value – hence, the new copy and the original pointer continue to point to the same data
- If operator= is not overloaded by the user, compiler provides a free one.
- Free operator= can make only a shallow copy
- If the constructor uses operator new, operator= should be overloaded
- If there is a need to define a copy constructor then operator= must be overloaded and vice-versa

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, in that we have already seen the how overloading works for the copy assignment operator and the same is also here and we had noted that this overloading is very important because unless you overload the compiler provides a free copy assignment which does only a shallow copy. So, in the presence of pointer variables that could give you a wrong semantics and you may need to overload and do a deep copy in the copy assignment operator.

Now in addition in blue I have shown what are the additional factors that you should keep in mind, that if you are constructing users operator new. When will the constructor use operator new when you have some dynamically allocable pointer members. So, when if you are using operator new, then your operator = should be overloaded. Because then you have a situation of deciding between shallow copy and deep copy. And at the same time if there is a need to define a copy constructor then there usually must be a need to define a copy '=' operator also and vice versa. So, while you do the design of classes please keep this point in mind because copy '=' operator is a very critical operator which needs to be overloaded.

(Refer Slide Time: 17:54)

Module 18
Partha Pratim Das

Objectives & Outcome
Motivation
Operator Function
Using global function
private data members
Using member function
operator overloading
Unary Operators
Summary

Program 18.04: Overloading Unary Operators

```
#include <iostream>
using namespace std;

class MyClass { int data;
public:
    MyClass(int d) : data(d) {}

    MyClass& operator++() { // Pre-increment:
        ++data; // Operate and return the operated object
        return *this;
    }

    MyClass operator++(int) { // Post-Increment:
        MyClass t(data); // Return the (copy of) object; operate the object
        ++data;
        return t;
    }

    void disp() { cout << "Data = " << data << endl; }

};

int main() {
    MyClass obj1(8);
    obj1.disp();

    MyClass obj2 = obj1++; // obj2.disp(); obj1.disp();
    obj2 = ++obj1;
    obj2.disp(); obj1.disp();

    return 0;
}
```

● Output
Data = 8
Data = 8
Data = 9
Data = 10
Data = 10

● The pre-operator should first perform the operation (increment / decrement / other) and then return the object. Hence its return type should be `MyClass`, and it should return `*this`.

● The post-operator should perform the operation (increment / decrement / other) after it returns the original value. Hence it should copy the original object in a temporary `MyClass t`; and then return `t`. Its return type should be `MyClass`.

NPTEL MOOCs Programming in C++ Partha Pratim Das 14

Just going for the unary operators can be overloaded the only reason I include them here is unary operators could be of two types prefix and postfix. So, this is a prefix operator and this is a postfix operator. I have already mentioned that the postfix operator is designated by an additional type in the signature, but this is just a placeholder this is just used by the compiler to resolve between the prefix and the postfix operator. Of course, you do not expect to pass a second parameter for a unary operator, because unary operator already has its parameter from the class on which the class on which this is invoked either it is this case or it is in this case.

Now, if you looking to the implementation and the return type. Which I would like to little bit highlight you to. So, if it is a prefix incrementor so what we know that it will first increment and return the object .Prefix operator should return the same object on which invocation has happened. So, that tells us that the same object has to come back and naturally return has to be a `*this`, which basically returns the same object. And before that the incrementation of the operation has already happened.

On the other hand, if you look into the post increment operator then, what you are supposed to get yes the operator has already worked the result of post increment is the original value. The original value that you had before the operator was invoked you get

that as a return value and then the object is actually incremented. So, it is done later on. So, if you have to since the object will get incremented the value in the object will get incremented after this operation any way. The value that is returned must be a copy of the original, because the original object is going to change as soon as you return from this operator.

So, now you need to copy this in terms of a temporary return that temporary and since you are returning a temporary as you have already discussed earlier your return type will have to be a returned by value it cannot be a return by reference. So, in terms of a post operator your return type will be a written by value type. So, these explanations I have written in here also in case you need to refer to them while you are preparing on this and that is how you can overload unary operators.

(Refer Slide Time: 20:28)

**Program 18.05: Overloading Unary Operators:
Pre-increment & Post Increment**

```
#include <iostream>
using namespace std;

class MyClass { int data; }

public:
    MyClass(int d) : data(d) {}

    MyClass operator++() { // Pre-Operator
        data *= 3;
        return *this;
    }

    MyClass operator++(int) { // Post-Operator
        MyClass t(data);
        data *= 3;
        return t;
    }

    void disp() { cout << "Data = " << data << endl; }

int main()
{
    MyClass obj1(10);
    obj1.disp();

    MyClass obj2 = obj1++;
    obj2.disp(); obj1.disp();

    obj2 = ++obj1;
    obj2.disp(); obj1.disp();
}
```

Output

- Data = 12
- Data = 12
- Data = 4
- Data = 8
- Data = 8

- The pre-operator and the post-operator need not merely increment / decrement
- They may be used for any other computation as this example shows
- However, it is a good design close to the native semantics of

Partha Pratim Das

Another point that I would like to highlight in terms of overloading not only unary, but many other operators as well; but I just show you with a unary operators that suppose this operator is unary increment operator, pre increment operator. But it does not necessarily mean that the functionality that you put in has to be an increment function. It could be some other functionality also, like here, I am showing that if I call this operator then my data field will get double and, but what is critical is the basic semantics of the

operator that it does return you the object on which you have invoked it. So, this will remain same, this will remain same, but the actual functionality could be different.

And the same holds for post operator as well I have given some different functionality that your data is divided by three. It does not matter, whatever you need, you can put there. But the basic points remain same that you need to copy the original object in a temporary and actually return that temporary through return by value, because that is what a post operator post incrimination operator will lead and these observations. So, it will be similar for the pre increment, pre decrement operator or all of the other unary operators and binary operators that you may have.

(Refer Slide Time: 21:47)

- Introduced operator overloading for user-defined types
- Illustrated methods of overloading operators using global functions and member functions
- Outlined semantics for overloading binary and unary operators

In this module, we have introduced the operator overloading for user-defined types. We have explained why it is important to overload the operators and we have illustrated the basic method of operator overloading by using global function and by using member functions of the classes. In the process, we also observed that you can use a friend function for overloading we will take that up in the next module. In this module, we have overall outlined the basic semantics of overloading for binary and unary operators.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 34
Overloading Operator for User - Defined Types: Part II

Welcome to module 19 of Programming in C++. All we have been discussing from the last module about Operator Overloading for User Defined Types.

(Refer Slide Time: 00:34)

The screenshot shows a presentation slide with a blue header bar. The title 'Module Objectives' is centered in the header. Below the header, there is a list of bullet points under the heading 'Objectives & Outline'. The slide also features a sidebar on the left with a vertical menu of topics. At the bottom, there is footer information including the NPTEL logo and the name 'Partha Pratim Das'.

Objectives & Outline

- Understand how to overload operators for a user-defined type (class)
- Understand the aspects of overloading by friend function and its advantages

NPTEL MOOCs Programming in C++ Partha Pratim Das

We have seen why operator overloading is important to create a specific complete types with possibilities of writing expression and creating algebra of different types. We have looked at how to overload operators using global functions and member functions. In this module we will take that forward, we will see that even with what we have proposed in terms of operator overloading using global and global functions and member functions there are some issues that crop up in operator overloading and we will show how friend function can be used for overloading operators properly.

(Refer Slide Time: 01:15)

This slide shows a module outline for operator overloading. The title is "Module Outline". The content includes a list of topics: Issues in Operator Overloading, Extending operator+, Overloading IO Operators, and Guidelines for Operator Overloading. On the left, there is a vertical navigation bar with links to Module 19, Partha Pratim Das, Objectives & Outline, Issues in Operator Overloading, Extending operators, Overloading IO Operators, and Guidelines Summary. On the right, there is a circular profile picture of Partha Pratim Das.

So, we will talk about issues in operator overloading and then specifically, we will discuss about extending operator '+' and about the IO operators these outline will be available as you know on the left hand side of your slides.

(Refer Slide Time: 01:32)

This slide is titled "Operator Function for UDT RECAP (Module 18)". It lists operator function options: Global Function, Member Function, and friend Function. It also lists Binary Operator, Unary Operator, and Examples. The examples table shows the expression, function, and remarks for each case. On the left, there is a vertical navigation bar with links to Module 19, Partha Pratim Das, Objectives & Outline, Issues in Operator Overloading, Extending operators, Overloading IO Operators, and Guidelines Summary. On the right, there is a circular profile picture of Partha Pratim Das.

Expression	Function	Remarks
a + b	operator+(a, b)	global / friend
++a	operator++(a)	global / friend
a + b	a.operator+(b)	member
++a	a.operator++()	member

Just a quick a recap from a the earlier module, there are three ways to overload operators,

these are the different operator functions that could not have for binary, these are the operator functions for unary and these are the different ways the invocation will actually happen based on an expression that we write this is how the functions will get invoked. We have already seen this.

(Refer Slide Time: 02:01)

The slide has a blue header with the title "Issue 1: Extending operator+" and a logo. The left sidebar lists navigation links: Module 19, Partha Pratim Das, Objectives & Outline, Issues in Operator Overloading, Extending operators, Overloading IO Operators, Guidelines, and Summary. The main content area contains bullet points and code examples:

- Consider a Complex class. We have learnt how to overload operator+ to add two Complex numbers:

```
Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;  
d3 = d1 + d2; // d3 = 4.1 +j 6.5
```
- Now we want to extend the operator so that a Complex number and a real number (no imaginary part) can be added together:

```
Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;  
d3 = d1 + 6.2; // d3 = 8.7 +j 3.2  
d3 = 4.2 + d2; // d3 = 5.8 +j 3.3
```

At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das". There is also a small circular video player showing a person speaking.

Now, let us talk about a extending or a designing. Let us go back to the consideration of the complex class. We have learnt how to overload operator ‘+’ to add two complex numbers so we can do this, d1 and d2 are two complex numbers we can add them we have seen how to write this using global function as well as using number function. Now what you want to do? We want to extend the power of this operator we are going to say that at times I want to add a real number with a complex number. A real number can be thought of as a complex number with a zero imaginary part.

So, could I extend this operator so that I can also write expressions like this, where d 1 is a complex value, where as this is a real value I would what to like to add that or, in a committed form there is a real value 4.2 and I want to add a complex number d 2 to that. Will my operator ‘+’ design be able to take care of this? If it can, then I really have a good value for my addition operator here. What we will try to show that what are the difficulties of doing this extension with the global function and member function and

how does a friend function help in this solution.

(Refer Slide Time: 03:20)

Module 19
Partha Pratim Das
Objectives & Outline
Issues in Operator Overloading
Extending operators
Overloading IO Operators
Guidelines
Summary

NPTEL MOOCs Programming in C++
Partha Pratim Das

● Consider a Complex class. Suppose we want to overload the streaming operators for this class so that we can write the following code:

```
Complex d;  
cin >> d;  
cout << d;
```

● Let us note that these operators deal with stream types defined in iostream, ostream, and istream:
● cout is an ostream object
● cin is an istream object

● We show why global operator function is not good for this
● We show why member operator function cannot do this
● We show how friend function achieves this

Partha Pratim Das

Then we will also take up a second issue that is of overloading IO operators. Again consider the complex class and we know for example if I write in parallelly in case of say a double type, then if I have a variable of type double then I can input from cin I can output to cout using these operators. Our intention is for the complex type also I should be able to do a similar kind of expression I should be able to write a similar kind of expression and functionality and that is basically will mean that the operators that we have the input operator and the output operator I should be able to overload them.

In this connection it will be good to note that whenever I do the IO if I am using cin, cin is actually an I stream object input stream object, whenever I do cout that actually is a stream object or out stream object. We will have to really design these operators so that we can overload keeping this stream types appropriately in our design. We will again show in this module that why global and member function solution for operator overloading fail in such cases and how the friend could help.

(Refer Slide Time: 04:52)

Program 19.01: Extending operator+ with Global Function

```
#include <iostream>
using namespace std;
class Complex { public: double re, im;
    explicit Complex(double r = 0, double i = 0): re(r), im(i) {} 
    void disp() { cout << re << " + " << im << endl; }
};
Complex operator+ (const Complex &a, const Complex &b) { // Overload 1
    return Complex(a.re + b.re, a.im + b.im);
}
Complex operator+ (const Complex &a, double d) { // Overload 2
    Complex b(d); return a + b; // Create temporary object and use Overload 1
}
Complex operator+= (double d, const Complex &b) { // Overload 3
    Complex a(d); return a + b; // Create temporary object and use Overload 1
}
int main(){
    Complex d1(2.6, 3.3), d2(1.6, 3.3), d3;
    d3 = d1 + d2; d3.disp(); // d3 = 4.1 +j 6.6
    d3 = d1 + 6.2; d3.disp(); // d3 = 8.7 +j 3.2
    d3 = 4.2 + d2; d3.disp(); // d3 = 5.8 +j 3.3
    return 0;
}
```

- Works fine with global functions - 3 separate overloading are provided
- A had solution as it breaks the encapsulation - as discussed in Module 18
- Let us try to use member function

• Note: A simpler solution uses Overload 1 and implicit casting (for this we need to make the constructor implicit). But that too breaks encapsulation. We discuss this when we take up operator+=.

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, let start by taking to extend operator plus with; first say, let say if I want to extent so by extension what we are trying to do we are trying to, let us looking to this code we are trying to add these two lines. We already have this we know how to add two complex numbers. We want to add a real with a complex. This is we are doing by global function so this overload one is the function that we had seen earlier which takes two complex numbers and add same component wise and returns a new complex number which is a result.

Now, if we want to do this next one which is where the first operand left operand is a complex number and the right operand is a real number then we will design a operator plus which has a pair of a parameters were the first one is complex and second one is double. This 6.2 will go as double, this will go as and that is my overload two, this is just another. Because we can overload as many as many types as we want as long as the overload resolution will get of them of C++ that we had discussed long ago can resolve between these overloads. If this particular version is taken then certainly we expect overload 2 to take place.

Similarly, if the last one is done were the first parameter the first argument, the left argument is a double then the third overload version should come in to invoke. In this

what we do? We do a sample thing we take this real number and construct a temporary complex number out of that real number which has zero imaginary part and you could just refer to the complex constructor you will be able to see how this is happening. Then we simply use the operator that we have, now we have two complex numbers so we just use the operator that we have already implemented in overload 1 to get them added has simple as that. So operator overload version 2 and 3 basically make sure that from the double argument we appropriately create a complex argument and then call the overload 1. With this certainly the problem gets solved three separate overloading solves the whole problem, but we know that since we are using global function we have all the nuances of breaking the encapsulation and that is not a very good preposition.

So, next what we will do we will try to use a member functions.

(Refer Slide Time: 07:44)

```
#include <iostream>
using namespace std;
class Complex { double re, im;
public:
    explicit Complex(double r = 0, double i = 0) : re(r), im(i) {}
    void disp() const { cout << re << " + " << im << endl; }
    Complex operator+ (const Complex &a) { // Overload 1
        return Complex(re + a.re, im + a.im); }
    Complex operator+ (double d) { // Overload 2
        Complex b(d); return *this + b; // Create temporary object and use Overload 1
    }
};

int main()
{
    Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;
    d3 = d1 + d2; d3.disp(); // d3 = 4.1 +j 6.5
    d3 = d1 + 6.2; d3.disp(); // d3 = 8.7 +j 3.2
    //d3 = 4.2 + d2; // Overload 3 is not possible - needs an object of left
    //d3.disp();
    return 0;
}

// Overload 1 and 2 works
// Overload 3 cannot be done because the left operand is double - not an object
// Let us try to use friend function

// Note: This solution too avoids the feature of cast operators
```

Let us move the encapsulation back, now these all have become private. Now we have moved to the operator overload functions here, overload one is here, exactly like the same way the only thing that has become different is the fact that it now has only the right hand side operand as argument to this function and as a parameter to this function left hand side operand is an object itself. If we talk about this particular case which adds two complex number we had seen this earlier it will invoke this overload 1. Similarly, in

this case where the argument is a single double number because the complex number on the left hand side operand is an object itself, so in this case it will invoke this overload 2, this is fine.

The issue starts with this third form. In the third form you can see there are two issues; one is certainly there are two operands and one of them is an object and the other one is a parameter to the member function to which you are making the invocation. So the consequent issue is, if I have to do that for 4.2 plus d 2 then the member function has to belong to the class of 4.2, because that is the left hand side operand where the invocation will happen. Now 4.2 is of double type which are built-in type and naturally I have no option of adding another operator to that or overloading the operators in the built-in type.

So there is no way that I can actually write for this, either expression we will not be able to support if I am using member functions for overloading my operators so that is the limitation of using the overloading with member functions because only some of the intended semantics overload 1 and 2 is what we have been able to support but the overload 3 will still fail, because we cannot write an appropriate function for that.

(Refer Slide Time: 10:04)

Module 19

Partha Pratim Das

Objectives & Outline
Issues in Operator Overloading
Extending operators
Overloading IO Operators
Guidelines Summary

Program 19.03: Extending operator+ with friend Function

```
#include <iostream>
using namespace std;
class Complex { double re, im; public:
    explicit Complex(double r = 0, double i = 0) : re(r), im(i) {}
    void disp() { cout << re << " + " << im << endl; }
    friend Complex operator+(const Complex &a, const Complex &b) // Overload 1
        return Complex(a.re + b.re, a.im + b.im);
}
friend Complex operator+(const Complex &a, double d) // Overload 2
    Complex b(d); return a + b; // Create temporary object and use Overload 1
}
friend Complex operator+(double d, const Complex &b) // Overload 3
    Complex a(d); return a + b; // Create temporary object and use Overload 1
}

int main()
    Complex d1(2.5, 3.2), d2(1.6, 3.3), d3;
    d3 = d1 + d2; d3.disp(); // d3 = 4.1 +j 6.6
    d3 = d1 + 6.2; d3.disp(); // d3 = 6.7 +j 3.2
    d3 = 4.2 + d2; d3.disp(); // d3 = 5.8 +j 5.3
    return 0;
}



- Works fine with friend functions - 3 separate overloading are provided
- Preserves the encapsulation too



Note: A simpler solution uses only Overload 1 and implicit casting (for this we need to or before constructor) will be discussed when we take up cast operators.



NPTEL MOOCs Programming in C++ Partha Pratim Das


```

So that is basic difficulty of what we are saying as issue one. So the solution is very

simple and elegant we have already discussed about friend function in depth. So all that we do is we go back to the solution of the global function which was the perfect solution, but the only difference being that the needed to expose the encapsulation. We know that the friend functions can actually look inside the class, so what we do? We go back to that solution and basically make the data members private as we would want. But move all the global functions into the scope of friend function in the class.

The same signature has the global function with prefixed `friend` and put inside the scope of the class and then we implement them. They just behave like the global functions only, but being friend they can access directly the private data members, so here, if we now have this particular function.

Now this the overload 2 is called where the second argument is a double and the overload 3 is also possible where the first argument is the double and depending on that, it gets resolved and now since it is a friend function it can access the internals of the complex class and actually it is possible to implement them even though the parameters `re` and `im` are private members. So friend gives us a very elegant solution for the whole a requirement and we will be able to achieve exactly what we needed without breaking the encapsulation that we were requiring earlier.

Just in the passing I would like to make a pointer to a note that I have put in here, is you will see that in this many of this solutions we have actually created three overloads which are very similar and related and these two overloads are primarily to convert the double in terms of the corresponding complex number and C++ does have a feature now known as the implicit casting which can do this task without explicitly being written by the user.

We have not yet studied about casting so when we studied that we will see that we will have even more compact solution where only overload 1 function would be able do all the three tasks without any difficulty. Right now I do not want that casting to happen and therefore, I have used a keyword `explicit` in front of the constructor which basically tell this compiler that do not do any casting for this object. So we will talk about this when we talk about the casting later on.

(Refer Slide Time: 12:58)

Overloading IO Operators:
operator<<, operator>> *

Module 19
Partha Pratim Das

Objectives &
Outline
Issues in
Operator
Overloading
Extending
operators
Overloading
IO Operators
Guidelines
Summary

Complex d1, d2;
cout << d1 << d2; // (cout << d1) << d2;

the signature of operator<< may be one of:

```
// Global function  
ostream& operator<< (ostream& os, const Complex &a);  
  
// Member function in ostream  
ostream& ostream::operator<< (const Complex &a);  
  
// Member function in Complex  
ostream& Complex::operator<< (ostream& os);
```

- Object to write is passed by constant reference
- Return by reference for ostream object is used so that chaining

NPTEL MOOCs Programming in C++ Partha Pratim Das

Let us move and talk about the second issue the IO Operators. Let us first try to understand issue what are we trying to do, there are two complex objects and I want to write this, this is the basic thing, I want to use this and be able to write them. Let us understand that the order in which these operators are applied are from left to right, so basically when I write this it means this. That is this operator, output operator is a binary one and it takes two operands; if you look into this the left hand side operand is a cout object and the right hand side operand is a d1 object. So I can say that in my signature how should this operator function look like the first operand is of the O stream type because cout is an object of the O stream type.

The second operand here is of the complex type because d1 is an object of the complex type. You will always have to work with signatures which match the desired use of the operator. Now you are left with the question of what should be the return type, what should it return, should it just return a void? No, if it returns a void then the difficulty is we will not be able write this. Because if we change the output operator then it means that we first output this then you output this. Now, you think about this instance of the output operator. This is the right hand side operand and this whole parenthesize things is the left hand side operand. So if this has to be there left hand side operand, but this is the result of a computation of the earlier invocation of the operator.

So, that result must be an output stream operator. So if I need to change and I would refer you to a recall the way we had discussed about operator ‘=’ we had similar discussion there as well that the return type of the this operator is decided by what is your operand in the input, because whatever you written here should be able to go back as input to the next invocation of the same operator. So that justifies that this is the signature that I need to ensure if I am implementing this operator as a global function. Of course, I can implement it as a operator as a member function. Now if I implement it as a member function naturally since there are two classes involved here O stream and complex I have two choices for member functions. One where it is a member in the O stream class and it takes complex as an input, or it is a member in the complex class and it takes O stream as an input. So, given that a let us try to see how the operator will get designed.

(Refer Slide Time: 16:09)

```
#include <iostream>
using namespace std;
class Complex {
public: double re, im;
    Complex(double r = 0, double i = 0): re(r), im(i) {}
};

ostream &operator<< (ostream& os, const Complex& a) {
    os << a.re << " +j " << a.im << endl;
    return os;
}

istream &operator>> (istream& is, Complex& a) {
    is >> a.re >> a.im;
    return is;
}

int main(){
    Complex d;
    cin >> d;
    cout << d;
    return 0;
}
```

- Works fine with global functions
- A bad solution as it breaks the encapsulation – as discussed in Module 18
- Let us try to use member function

NPTEL MOOCs Programming in C++ Partha Pratim Das

Let us try to implement it with the overloading the IO operator using as a global function. So this is the basic overloading. The signature we have already discussed so what we need to do in terms of the actual output we will just need to take this two components and print them with a ‘+ j’ for the complex number notation and that should achieve my output operation and since I have to written the same output stream operator so whatever I get in as a parameter here must be returned as a value of this operator computation. In terms of this I want to use the complex number for printing certainly we

could have used a call by value here, but that would mean an unnecessary copy so we are using a reference parameter and as it is a convention we do not expect the output operator to change the value therefore it is a constant reference parameter.

Similarly, I can write I stream operator that is input streaming operator everything look very similar to that only difference being now the complex number that comes in as a parameter has to be a non constant one, because I am doing an input. So with the input I expect that complex number to change. I do the component wise input and the remaining discussion remain the same that.

Finally, I return the input stream that I have got as a first parameter. If you do this then we pretty much have input output for the complex number and we are able to write this and it will work exactly as it does for any int or double types. So the only difficulty in this solution is it breaks the encapsulation, as we pretty much know by now that any overloading with global function will break the encapsulation.

(Refer Slide Time: 18:06)

Overloading IO Operators with Member Function

Module 19
Partha Pratim Das

Objectives & Outline
Issues in Operator Overloading
Extending operators
Overloading IO Operators
Guidelines & Summary

- Case 1: operator<< is a member in ostream class:

```
ostream& ostream::operator<< (const Complex &a);
```

This is not possible as ostream is a class in C++ standard library and we are not allowed to edit it to include the above signature
- Case 2: operator<< is a member in Complex class:

```
ostream& Complex::operator<< (ostream& os);
```

In this case, the invocation of streaming will change to:

```
d << cout; // Left operand is the invoking object
```

This certainly spoils the natural syntax

- IO operators cannot be overloaded by member functions
- Let us try to use friend function

NPTEL MOOCs Programming in C++ Partha Pratim Das

So let us take the same step let us try to do this by member function. Now if we want to do this by member function certainly we have noted there are two possibilities that the case one is, operator output say is a member of O stream class. So we will have a

signature like this. Now this is not possible. Why is this not possible? Because O stream is a not a class that your I have written, O stream is a class which is provided as a part of the C++ standard library and we are not allow to add members there, we are not allowed to edit that class in any way. So if we cannot edit that we cannot add a new member function like this. So this possibility is ruled out.

I can have a second option is, complex is my class so I can certainly try to overload the operator output in terms of a member function in the complex class so then it will look like something like this, because since I am overloading in the complex class my default parameter is a complex object so all that I need to specify is a O stream operator. But this is has a severe consequence. The consequence is my earlier order was O stream object and then the complex object this was the left right order, but here since complex is the class on which this is a member the order will become the complex object and O stream object because this has now become the right hand of the operand. So which means that the output done with this kind of an operator if I do then it will have to be written as in this notation `d << cout` not `cout << d` the way we are used to understanding this operator.

This discussion will tell you that it is actually not possible to overload the IO operator using member function either in the O stream class or in the user defined type.

(Refer Slide Time: 20:13)

The screenshot shows a presentation slide titled "Guidelines for Operator Overloading". The slide is part of "Module 19" by Partha Pratim Das. The navigation bar includes links for "Objectives & Outline", "Issues in Operator Overloading", "Extending Operators", "Overloading IO Operators", "Guidelines", and "Summary". The main content area contains a bulleted list of guidelines:

- Use global function when encapsulation is not a concern. For example, using `struct String { char* str; }` to wrap a C-string and overload `operator+` to concatenate strings and build a String algebra
- Use member function when the left operand is necessarily a class where the operator function is a member and multiple types of operands are not involved
- Use `friend` function, otherwise
- While overloading an operator, try to preserve its natural semantics for built-in types as much as possible. For example, `operator+` in a Set class should compute union and NOT intersection
- Usually stick to the parameter passing conventions (built-in types by value and UDT's by constant reference)
- Decide on the return type based on the natural semantics for built-in types. For example, as in pre-increment and post-increment operators
- Consider the effect of casting on operands
- Only overload the operators that you may need ([minimal design](#))

A small video player window in the bottom right corner shows a person speaking, identified as Partha Pratim Das.

So we are left with only one option that is to use the friend function I have not illustrated that here because you all know, all that you will need to do is to take the global function and move it inside the class and prefix them with the friend and that will simply solve the problem. Before I close I would like to leave you with few guidelines for operator overloading, because we have discussed in different context the overloading actions operator overloading by global function, by member function and friend function. If you are overloading with the global function you will do that provided encapsulation is not a concern that you really do not care about encapsulation, and typically that will happen if you are using structures only.

Like the string example, C-string example we had shown where we are using a structure string just to wrap the char* pointer, just to wrap the C-string and then having done that we are able to write and overload for operator + which can add or concatenate two string object or one string object and one char* pointer and so on. When you have no concern for the encapsulation or little concern for the encapsulation use global functions for operator overloading

Similarly, use member function, when it is guaranteed that the left operand is necessarily a class where you can make it a member. So, we have seen situations where it fails, but if in the complex number design the operator+ if we did not have the requirement of being able to add a real number with a complex value by operator+ then we can pretty much keep the operator+ overloaded as a member function of complex.

But in other cases where the operands might be a multiple types or it is not possible that you can make the appropriate left operand of a type where you can add the member function you will not be able to use that. So whenever possible use member functions as for operator overloading and in all the remaining cases and the major cases we have already seen and that you will find that lot of operators are overloaded in that same style you will have to use the friend function for overloading the operators.

Further to this I would also like to highlight that whenever you overload an operator basically you are writing a function, so you are preserving the arity, the associativity, the precedence, you are basically trying to create a new function and you are associating that

function with the operator symbol. Now most of us, the programmers have been accustomed to certain semantics for the operator symbols, we are used to thinking in a certain way. For example, if I see an operator plus then in some way I get a sense of adding things putting things together making a union and things like that. If you are using operator+ for a certain type, you should look for does this the semantics that you are putting in the overload of the operator+ does it sound similar to doing some kind of a union addition kind of things. For example, if you are overloading operator+ for a set class then a certainly you should use that to compute union and not to compute intersection.

Those kind of things that a naturality, the more you make the operator overloading for your type closing semantics to the behavior of the operator for the built-in types you will find that your class will behave more nicely and other programmers would be able to understand and debug your class in a much better way they will be able to use your class in a much better way.

Similarly, for passing the parameters to your operator function try to follow strict to the default parameter passing the invention which is, as I had mentioned several times is if there is a built-in type you pass it by value and if there is a user defined type then you try to pass it by constant reference. Certainly you will have to be careful about the requirements for example, we just saw that if we are doing an input streaming operator then we need to pass the parameter the data value as a reference which is non constant because we expect that the input streaming we will actually put value into that, but otherwise first attempt would be to try to honor the parameter passing convention.

In terms of the return type again you try to rely more on the natural semantics of the built-in types for example, we just discuss that the input direction and the output direction, input streaming and output streaming operators have been designed with chaining so you would preserve that property when you overload the input -streaming and output-streaming operators. They should be chainable you should not or write it in a way so that I cannot chain the output stream or the input stream. And therefore, that has dictated that the return type of this operator has to be same as the left hand side operand of the operator.

And that is what I mean by the natural semantics of the built-in type or another example of natural semantics is as we discussed in case of pre-increment and post-increment cases. For pre-increment for it is built-in type it is a original object which actually gets increment and then object comes back so you should use a return by a reference as a refer post increment you get an old value of the object so it will have to be a new object. Therefore, you cannot get it by reference you will have to get it by value so please be careful about these decisions. Then the next is you will have to consider the effect of casting on the operands, certainly since we have not discussed about casting I am not in a position to elaborate more on this, but while you discuss casting we will refer back on this. But please do remember that the casting makes a lot of impact on the way you overload your operators.

Finally, I will have a strong advice that please, I mean once you get a custom to overloading operators I have often seen particularly the young programmers they try to overload all operators that are available to be overloaded. And makes a type which has 30 overloaded operators, but in practice may be only 5 or 6 of them are used. So, please while you are making the overload make sure that you make a design which is minimal. Minimal design is a basic requirement of any good programming. And you only overload those operators that you really need actually need to build up your type to build up your total algebra.

(Refer Slide Time: 27:45)

The screenshot shows a presentation slide titled "Module Summary". On the left, there's a vertical sidebar with a logo at the top, followed by a list of topics: Module 19, Partha Pratim Das, Objectives & Outcome, Issues in Operator Overloading, Extending operators, Overloading IO Operators, Guidelines, and Summary. The main content area has a blue header bar with the title "Module Summary". Below the header, the slide lists several bullet points under the heading "Guidelines for Operator Overloading":

- Several issues of operator overloading have been discussed
- Use of friend is illustrated in versatile forms of overloading with examples
- Discussed the overloading of I/O (streaming) operators
- Guidelines for operator overloading are summarized
- Use operator overloading to build algebra for:
 - Complex numbers
 - Fractions
 - Strings
 - Vector and Matrices
 - Sets
 - and so on ...

At the bottom of the slide, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das". To the right of the slide, there is a circular video player showing a man (Partha Pratim Das) speaking.

So please keep these guidelines in mind and then your design of operators should really turn out to be good. In this module we have actually continued from our previous module, the module 18 and we have discussed about several issues of operator overloading specifically the issue of doing operator overloading when your parameter types could be varied and particularly in terms of I/O when your two parameters are of two different class types and so on. And we have shown that using the friend function there are really elegant nice solutions to operator overloading, so I will also refer back to our discussion on the friend function module where I had mentioned the three situations under which the friend should be used as a function and this was a third situation where you really need to use friend, and you can now realize has to the importance of friend in doing any nice design of operator overloading.

We have also met some guidelines for you, in terms of how you should overload your operators or which operators you should be overloading and so on, please keep those in mind. And finally, I would advise that in module 18 when we started, we started with a motivation of building complete types, complete algebra for variety of different arithmetical types and others, so a complex number, fractions, string, vector matrices and so on. So I will urge that you practice on the same lines, you practice and start building complete type.

For example, a good exercise would be to try to build a complex type in full which supports all operators that say the int type supports, and it may be actually complex will needs some more operators like you need to find the absolute value of a complex number, the norm of a complex number, you need to find the complex conjugate of a complex number. So, you will have to identify proper operators and overload them. And in that process you should be able to have a complete complex type which would behave exactly like your int type and you will able to write expressions of your complex type values and variables as you do in case of int type.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 35
Namespace

Welcome to Module 20 of Programming in C++. In this module we will talk about Namespaces.

(Refer Slide Time: 00:28)

The screenshot shows a presentation slide titled "Module Objectives". The slide has a dark blue header bar with the title and a white content area. On the left, there is a vertical sidebar with a logo at the top and a list of navigation links. The main content area contains a single bullet point under the heading "Objectives & Outline".

Module Objectives

Module 20
Partha Pratim Das

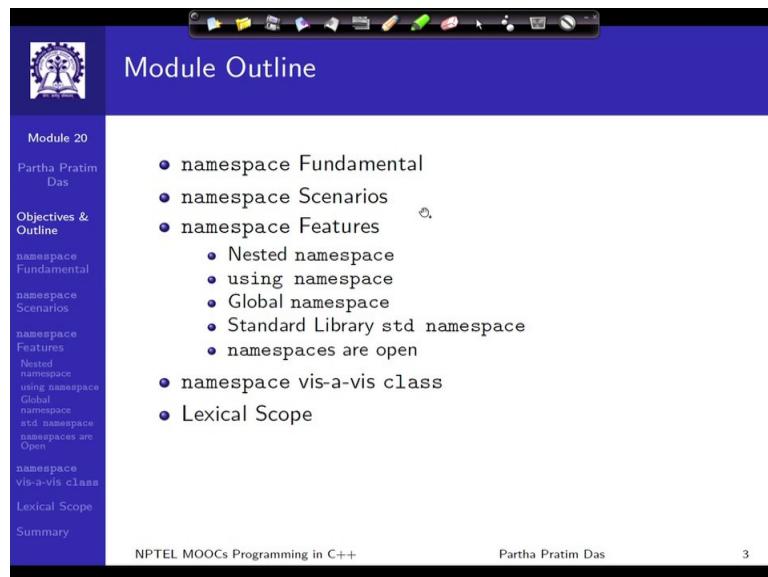
Objectives & Outline

- Understand namespace as a free scoping mechanism to organize code better

NPTEL MOOCs Programming in C++ Partha Pratim Das 2

Namespaces are a concept of lexical scoping, of which we have variety of options that you already know in C++. But, we will expose you to this additional concept of scoping and how does it help in organising code structure.

(Refer Slide Time: 00:51)



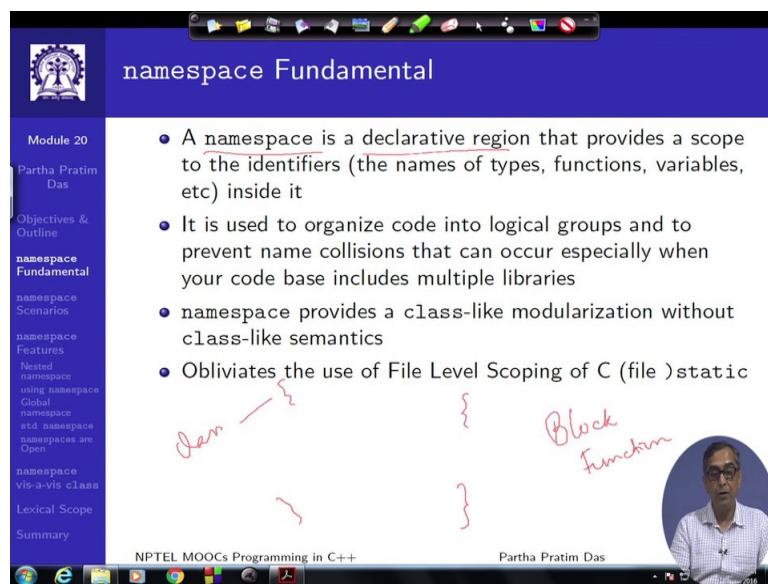
This slide shows the 'Module Outline' for Module 20, taught by Partha Pratim Das. The outline includes:

- namespace Fundamental
- namespace Scenarios
- namespace Features
 - Nested namespace
 - using namespace
 - Global namespace
 - Standard Library std namespace
 - namespaces are Open
- namespace vis-a-vis class
- Lexical Scope

The sidebar lists topics such as Objectives & Outline, namespace Fundamental, namespace Scenarios, namespace Features, Nested namespace, using namespace, Global namespace, std namespace, namespaces are Open, namespace vis-a-vis class, Lexical Scope, and Summary.

This is the outline, and you will find it on the left of the every slide as we go forward.

(Refer Slide Time: 01:00)



This slide is titled 'namespace Fundamental'. It contains a bulleted list of points and handwritten annotations:

- A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it
- It is used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries
- namespace provides a class-like modularization without class-like semantics
- Obliviates the use of File Level Scoping of C (file)static

Handwritten annotations include: 'Declarative region' with arrows pointing to the first point, curly braces around the last two points, and 'Block function' with curly braces around the last point. A circular video player window shows a video of Partha Pratim Das.

So, first let me introduce what is a namespace. A namespace, as I said is a declarative region; it is a scope, like this. So, we already are aware of scope like block, we know that every function has a scope; we know class has a scope, class some, class name has a

scope. So, namespace is something absolutely similar to that. Here we have a scope it is a declarative region, within which I can have variety of different identifiers, variety of different symbols. We have types, functions, variables, classes, and other namespaces and so on.

The major purpose as I said is to organise the coding into logical groups. And that is a that is a very critical requirement. And, one of the major reasons that namespace exists is to prevent name clash, name collision that can occur. Especially, when the code base include multiple libraries, the code base intends to use names which are reserved by the third party libraries or standard library, or code base evolves has been developed by independent developers and inadvertently the same set of names have got used. So, this is the main purpose off, this is what namespace is, it defines a scope and this is the main purpose to organise the code. Namespace provides a class like modularization, like we know that every class definition is kind of a modular definition, but the difference is it does not have any semantics. It is just a scoping rule, class also does scoping, but it does scoping with class semantics. Namespace separately will not have any semantics.

And, for those of you who are familiar with file scope in C, like which we say is a file static variables or file static functions, use of namespace will obviate the requirement of file scope, as is used in C. So, if you have been using the same, then whenever you need that, you should actually not use this and rather use namespace in that place.

(Refer Slide Time: 03:33)

```
#include <iostream>
using namespace std;

namespace MyNameSpace {
    int myData; // Variable in namespace
    void myFunction() { cout << "MyNameSpace myFunction" << endl; } // Function in namespace
    class MyClass { int data; // Class in namespace
public:
    MyClass(int d) : data(d) {}
    void display() { cout << "MyClass data = " << data << endl; }
};
int main() {
    MyNameSpace::myData = 10; // Variable name qualified by namespace name
    cout << "MyNameSpace::myData = " << MyNameSpace::myData << endl;
    MyNameSpace::myFunction(); // Function name qualified by namespace name
    MyNameSpace::MyClass obj(25); // Class name qualified by namespace name
    obj.display();
    return 0;
}
```

Let me introduce you to the actual code as to how to write this, this is not otherwise a meaningful example. So here, I am saying namespace is a keyword with which like the class keyword you do and that should be followed by the namespace name. So, the syntax is pretty much like how you define a class. It has an associated scope in terms of matching curly braces, and anything that you write within that namespace is belongs to the namespace, which means that any symbol that you write inside this gets qualified by the name of the namespace.

So, here I show 3 different kinds of entries. One is a name of a variable, my data, one is a function and one is a class, myData, myFunction, MyClass. And then, in the main I show how to use that. If I want to use the data, then I need to write it in this form. So, you can see that the variable name was my data that has got qualified by the namespace name. This is exactly how class names qualify, for example, I would immediately remind you about static data members in classes this is how you write the static data members in classes. So, the namespace name followed by the symbol name, separated by the scope resolution operator is a way to refer to the namespace object. For example, for the function that we have above, this name is, for the class, this name. In this context of the namespace, here if I simply write myFunction and try to invoke that then I will get compilation error. Because, there is nothing called no symbol called my function in this

program. The, my function symbol is within the namespace and therefore will always have to be prefixed with the namespace. So, this is a basic way you define namespaces and you use namespaces.

(Refer Slide Time: 05:43)

Module 20
Partha Pratim Das

Objectives & Outline

- Namespace Fundamental
- Namespace Scenarios
- Namespace Features
- Nested namespace using namespace
- Global namespace
- std namespace
- Namespaces are Open
- Namespace vis-a-vis class
- Lexical Scope
- Summary

Scenario 1: Redefining a Library Function (Program 20.02)

● cstdlib has a function int abs(int n); that returns the absolute value of parameter n
 ● You need a special int abs(int n); function that returns the absolute value of parameter n if n is between -128 and 127. Otherwise, it returns 0
 ● Once you add your abs, you cannot use the abs from library! It is hidden and gone!
namespace comes to your rescue

Name-hiding: abs()	namespace: abs()
<pre>#include <iostream> #include <cmath> int abs(int n) { if (n < -128) return 0; if (n > 127) return 0; if (n < 0) return -n; return n; } int main() { std::cout << abs(-203) << " " << abs(-6) << " " << abs(77) << " " << abs(179) << std::endl; // Output: 0 6 77 0 return 0; }</pre>	<pre>#include <iostream> #include <cmath> namespace myNS { int abs(int n) { if (n < -128) return 0; if (n > 127) return 0; if (n < 0) return -n; return n; } } int main() { std::cout << myNS::abs(-203) << " " << myNS::abs(-6) << " " << myNS::abs(77) << " " << myNS::abs(179) << std::endl; // Output: 0 6 77 0 std::cout << abs(-203) << " " << abs(-6) << " " << abs(77) << " " << abs(179) << std::endl; // Output: 203 6 77 179 return 0; }</pre>

NPTEL MOOCs Programming in C++ Partha Pratim Das 6

Let us look at two scenarios, one simple and one little bit more involved. Here is the scenario where I am trying to redefine the library function. So, all of us know that library function has an ABS function, standard library has a ABS function which finds the absolute value, but I want to give a different behaviour to that. I want to say that within -128 and 127 it will find the absolute, but otherwise if it is outside that range then it will return a zero. So, the simple way to do this, I define the abs function and start using it, and the flip side of this is if we do that then, the abs function that exists in the C standard library is hidden, that is once I have defined my abs function, then the abs function that is available from the library is no more available. So if I just use abs, it will always mean my abs it will never mean the abs that existed in the library. So, I lose by this if I do this in C and of course, then we can do it in C++ I lose the ability to actually refer to the original library function.

So, namespace provides a nice solution to that. If I want to do this and also want to keep referring to the library function, I can still define my abs function, but I will just put it in

a new namespace. So, I have given the name of that as myNS. So, with that if I refer to the abs function as my NS :: abs, then it refers to this function. But if I just say abs, since there is no abs available because this particular new version of abs is within myNS name scope. So, if I just say abs then it means the abs of the one that exists in the standard library. In this way I can protect my new introduced definitions without clashing with the library name that exists. This is one very typical requirement that you will often face and this is a solution using namespace is a good solution.

(Refer Slide Time: 08:04)

Module 20
Partha Pratim Das
Objectives & Outline
namespace Fundamental
namespace Scenarios
namespace Features
Nested namespace
using namespace Global
using namespace std
namespaces are Open
namespace vis-a-vis class
Lexical Scope Summary

Scenario 2: Students' Record Application:
The Setting (Program 20.03)

- An organization is developing an application to process students records
- class St for Students and class StReg for list of Students are:

```
#include <iostream>
using namespace std;
class St { public: // A Student
    typedef enum GENDER { male = 0, female } ;
    char *n, GENDER g : name(strcpy(new char[strlen(n) + 1], n)), gender(g) {};
    void setRoll(int r) { roll = r; } // Set roll while adding the student
    GENDER getGender() { return gender; } // Get the gender for processing
    friend ostream& operator<< (ostream& os, const St& s) { // Print a record
        cout << ((s.gender == St::male) ? "Male" : "Female")
            << s.name << " " << s.roll << endl;
        return os;
    }
private:
    char *name; GENDER gender; // name and gender provided for the student
    int roll; // roll is assigned by the system
};
class StReg { // Students' Register
    St **rec; // List of students
    int nStudents; // Number of student
public:
    StReg(int size) : rec(new St*[size]), nStudents(0) {}
    void add(Students* s) { rec[nStudents] = s; s->setRoll(++nStudents); }
    Students *getStudent(int r) { return (r == nStudents + 1) ? 0 : rec[r - 1]; }
};
```

- The classes are included in a header file Students.h

NPTEL MOOCs Programming in C++ Partha Pratim Das 7

Let me quickly run through a development scenario. This is what very regularly happens in any organisation. Suppose, an organisation is developing an application to process students' records and let us say there are two classes; one for the students. So, we have two classes; this represents a student and this represents a list of students. You may not really bother about the details though it actually is a correct program. So, you could at a later point of time read through and actually find what this program is doing. It certainly has different, it has the constructor, it has some set and get functions and it has a output operator to be able to write a student record. And, this has a list and it keeps track of the number of students that exist and it has an option to add a student to this list. So, whenever you add a student the roll number of that student gets allocated. In terms of developing this processing application, what the organisation does and that is very

typical that somebody senior of the designers will possibly have designed this classes. And then the task is given to multiple developers to develop different parts of the system. So, these classes are designed and given in a header file, Students. h.

(Refer Slide Time: 09:28)

Module 20

Partha Pratim Das

Objectives & Outline

namespace Fundamental

namespace Scenarios

namespace Features

Nested namespace

using namespace Global namespace

std namespace namespaces are Open

namespace vis-a-vis class

Lexical Scope

Summary

**Scenario 2: Students' Record Application:
Team at Work (Program 20.03)**

- Two engineers – **Sabita** and **Niloy** – are assigned to develop processing applications for male and female students respectively. Both are given the **Students.h** file
- The lead **Purnima** of Sabita and Niloy has the responsibility to integrate what they produce and prepare a single application for both male and female students. The engineers produce:

Processing for males by Sabita <pre>#include <iostream> using namespace std; #include "Students.h" extern StReg *reg; void ProcessStudents() { cout << "MALE STUDENTS: " << endl; int r = 1; St *s; while (s = reg->getStudent(r++)) if (s->getGender() == St::male) cout << *s; cout << endl << endl; return; } //////////////// Main.cpp /////////////////// #include <iostream> using namespace std; #include "Students.h" StReg *reg = new StReg(1000); int main() { St s("Partha", St::male); reg->add(&s); ProcessStudents(); return 0; }</pre>	Processing for females by Niloy <pre>#include <iostream> using namespace std; #include "Students.h" extern StReg *reg; void ProcessStudents() { cout << "FEMALE STUDENTS: " << endl; int r = 1; St *s; while (s = reg->getStudent(r++)) if (s->getGender() == St::female) cout << *s; cout << endl << endl; return; } //////////////// Main.cpp /////////////////// #include <iostream> using namespace std; #include "Students.h" StReg *reg = new StReg(1000); int main() { St s("Ramala", St::female); reg->add(&s); ProcessStudents(); return 0; }</pre>
--	---

NPTEL MOOCs Programming in C++

Partha Pratim Das

8

At this point the responsibility is given to; let us say to engineers, by dividing the development, separately for male students and for female students. They will have different requirements of hostel, different requirements of subject classification and so on. So, this is what is being done by Savitha, for male students; similar development being done by Niloy, for female students; and Poornima is their lead, who finally has to integrate both these applications into your final applications. So what happens is, it may be incidental, coincidental, whatever, is Savitha calls this whole processing application that she is doing for the male students as process student; and, Niloy also chooses the same name of the function. And then, they write independent main applications to test out whatever they have developed. You need not bother about what this is doing, this basically is, if this application prints the male students, this application prints the female students, but that is just an indicative one I am not really concerned about what the processing is happening. But the fact is that, they have chosen incidentally the same function name and independently developed it together. Now, both of them submit the code to Poornima, the lead, now Poornima has to put them together.

(Refer Slide Time: 10:56)

The slide has a dark blue header with the title. On the left, there's a sidebar with a logo and a vertical list of topics under 'Module 20'. The main content area contains a code snippet and a bulleted list of points.

Module 20

- Partha Pratim Das
- Objectives & Outline
- namespace Fundamental
- namespace Scenarios
- namespace Features
- Nested namespace
- using namespace
- Global namespace
- std namespace
- namespaces are Open
- namespace vis-a-vis class
- Lexical Scope
- Summary

**Scenario 2: Students' Record Application:
The Integration Nightmare (Program 20.03)**

To integrate, Purnima prepares the following main() in her Main.cpp where she intends to call the processing functions for males (as prepared by Sabita) and for females (as prepared by Niloy) one after the other:

```
#include <iostream>
using namespace std;
#include "Students.h"

void ProcessStudents(); // Function from App1.cpp by Sabita
void ProcessStudents(); // Function from App2.cpp by Niloy

StReg *reg = new StReg(1000);

int main() {
    St s1("Ramala", St::female); reg->add(&s1);
    St s2("Partha", St::male); reg->add(&s2);

    ProcessStudents(); // Function from App1.cpp by Sabita
    ProcessStudents(); // Function from App2.cpp by Niloy

    return 0;
}
```

- But the integration failed due to name clashes
- Both use the same signature void ProcessStudents(); for their respective processing function. Actually, they have several functions, classes, and variables in their respective development with the same name and with same / different purposes
- How does Purnima perform the integration without major changes in the codes? – namespace

NPTEL MOOCs Programming in C++ Partha Pratim Das 9

So, certainly, if we just refer back once, Poornima will neither use this main, nor will use this main, because she has to integrate both of this codes. So, she will have to write a main unified application of for all. So, she tries to write that. So, she has to call these functions, the process students' functions. She has to put their definitions on top and she finds that, both of them use the same function name. So, if she would copy their codes, then this is what it will look like this is the same. As you can understand that this will give a name clash and it cannot be same, it will simply not compile. Now, whole integration process has failed.

I have shown in terms of a function name just as a indicative one, but in reality, certainly, the application would have would be in couple of thousands of lines. There could be several symbols, several functions, global variables, class names, several types and so on, which may have the same name between the two developers. There could be more developers also. There could be that in some cases, the names that are same between two developers' programs or code means the same thing; in some cases, same name as used by Savitha, will be used by Niloy in a different meaning. So, this is a very very difficult problem. If Poornima has to integrate, she has to understand the whole code and then edit that make changes within that and that will become a complete nightmare. So, this is what is known as the integration nightmare that typically happens in an organisation.

But, the other side of the fact is independently both the applications as developed by Savitha for the male students and by Niloy for the female students independently they work, it's only that the, they are sharing certain common names and therefore namespace turns out to be a panacea which can solve this problem very easily.

(Refer Slide Time: 13:15)

```

Module 20
Partha Pratim Das
Objectives & Outline
namespace Fundamental
namespace Scenarios
namespace Features
Nested namespace
using namespace Global
namespace
namespace Open
namespace vis-a-vis class
Lexical Scope
Summary

Scenario 2: Students' Record Application:
Wrap in Namespace (Program 20.03)

Processing for males by Savitha
App1.cpp
#include <iostream>
using namespace std;
#include "Students.h"
extern StReg *reg;
namespace App1 {
void ProcessStudents() {
cout << "MALE STUDENTS: " << endl;
int r = 1;
St *s;
while (s = reg->getStudent(r++))
if (s->getGender() == St::male)
cout << *s;
cout << endl << endl;
return;
}
}

Processing for females by Niloy
App2.cpp
#include <iostream>
using namespace std;
#include "Students.h"
extern StReg *reg;
namespace App2 {
void ProcessStudents() {
cout << "FEMALE STUDENTS: " << endl;
int r = 1;
St *s;
while (s = reg->getStudent(r++))
if (s->getGender() == St::female)
cout << *s;
cout << endl << endl;
return;
}
}

NPTEL MOOCs Programming in C++
Partha Pratim Das
10

```

All that Poornima now has to do is to take the applications that Savitha had developed and Niloy had developed and put them into two different namespaces. She decides on two names, app 1 and app 2 and puts Savitha's developments in this, puts Niloy's developments in this. So, once that has been done, then within this application, it is all within the same namespace. So, the application still works. This part will still work, but when you are outside when you are looking at from main functions point of view these two are into two different namespaces. So, they are basically, different functions.

(Refer Slide Time: 14:01)

The slide title is "Scenario 2: Students' Record Application: A Good Night's Sleep (Program 20.03)". On the left, there is a sidebar with a logo and a list of topics: Module 20, Partha Pratim Das, Objectives & Outline, namespace Fundamental, namespace Scenarios, namespace Features, Nested namespace, Using namespace, Global namespace, std namespace, namespaces are Open, namespace vis-a-vis class, Lexical Scope, and Summary. The main content area contains the following code:

```
Now the integration gets smooth:  
#include <iostream>  
using namespace std;  
  
#include "Students.h"  
  
namespace App1 { void ProcessStudents(); } // App1.cpp by Sabita  
namespace App2 { void ProcessStudents(); } // App2.cpp by Niloy  
  
StReg *reg = new StReg(1000);  
  
int main() {  
    St s1("Ramala", St::female); reg->add(&s1);  
    St s2("Partha", St::male); reg->add(&s2);  
  
    App1::ProcessStudents(); // App1.cpp by Sabita  
    App2::ProcessStudents(); // App2.cpp by Niloy  
  
    return 0;  
}  
  
Clashing names are made distinguishable by distinct namespace names
```

At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das". The slide number is 11.

So, then she will come back after putting encapsulating them into namespaces. Now, she is back into the integration. This is, Savitha's applications is now quite processing students function in app 1 namespace; for Niloy, that is in the app 2 namespace. So, all that she needs to do is, by doing this, she has resolved the clash of name between the two development units and in terms of the application, all that she needs to do is to use the namespace prefix and call the two functions independently, one after the other or whatever she wants to do.

So, this is a very typical approach, which can solve a huge amount of practical integration problem and that needs to be kept in mind. Of course, it is not a good idea that multiple developers in the same system will not be able to coordinate and resolve to have distinct names, but it often is a good idea to use namespace also to make different modules so that in between different modules you will not have to really bother about what names, what different supporting functions, supporting classes and all that, you may be using. So, if we are doing a student record development, there could be one module which deals primarily with the students' academic part; one module deals with the students fees; one module deals with the students hostel; another module will deal with their timetable and so on. So, a good way to organise the code and to the design

would be to assign different namespaces to this modules and basically separate out the code in that manner, so that you make sure that it will never clash among themselves.

(Refer Slide Time: 15:43)

Module 20
Partha Pratim Das

Objectives & Outline
namespace Fundamental
namespace Scenarios
namespace Features
Nested namespace
using namespace Global namespace std namespaces are Open
namespace vis-a-vis class
Lexical Scope
Summary

Program 20.04: Nested namespace

- A namespace may be nested in another namespace

```
#include <iostream>
using namespace std;

int data = 0; // Global name ::

namespace name1 {
    int data = 1; // In namespace name1
    namespace name2 {
        int data = 2; // In nested namespace name1::name2
    }
}

int main() {
    cout << data << endl; // 0
    cout << name1::data << endl; // 1
    cout << name1::name2::data << endl; // 2

    return 0;
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 12

So, let me just to complete the discussion you have understood why namespace is important. So, let me just go over the quick features that a namespace has. As I said, it is very similar to class; like class, the namespaces can be nested. So, I have a namespace name1 here and within that I have another namespace. So, all that happens is, if I nest one namespace into another, the name of this nested namespace is qualified by the outer namespace. So, that is the simple thing. So, which means that this data is in the namespace name1 whereas, this data is in the namespace of name1 :: name 2, because this itself is name 1 :: name 2. So, if we write this code, if I just write data then, data is only outside so it will mean this data. If I write name 1 :: data it means this data, if I write name1 :: name 2 :: data, it means this data. So, as you keep on nesting namespaces, more and more prefix get added. So, any level of nesting is possible for namespaces.

(Refer Slide Time: 16:56)

Module 20

Partha Pratim Das

Objectives & Outline

namespace Fundamental

namespace Scenarios

namespace Features

Nested namespace

using namespace

Qualified namespace

std namespace

namespaces are Open

namespace vis-a-vis class

Lexical Scope

Summary

Program 20.05: Using using namespace and using for shortcut

- Using using namespace we can avoid lengthy prefixes

```
#include <iostream>
using namespace std;

namespace name1 {
    int v11 = 1;
    int v12 = 2;
}

namespace name2 {
    int v21 = 3;
    int v22 = 4;
}

using namespace name1; // All symbols of namespace name1 will be available
using name2::v21; // Only v21 symbol of namespace name2 will be available

int main() {
    cout << v11 << endl; // name1::v11
    cout << name1::v12 << endl; // name1::v12
    cout << v21 << endl; // name2::v21
    cout << name2::v21 << endl; // name2::v21
    cout << v22 << endl; // Treated as undefined
    return 0;
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

Often, if you do it like this then it might become difficult that as you have multiple nestings and multiple symbols that you want to use, every time you will have to put the namespace name. So, there is a shortcut that you can use. There are basically two shortcuts that you can use; one is these are called using. So, one shortcut that you can do is, you say using and then you put a name of a namespace. So, you say using namespace name 1. What this will mean is any symbol that is subsequently used after this; you will try to check if this namespace name 1 has that symbol. And if it has that symbol then you will be referring to that symbol. So, this is the feature of using namespace.

The other using feature is, you can say using and you can say that, actually a qualified symbol itself. So, if you specify a qualified symbol, name qualified symbol then whenever later on you just talk about the symbol name, it will mean this qualified symbol. So, this example should help. So, I have two namespaces, name1 and name2; name1 has two symbols; name2 as two symbols; and, I have a using on name1, the namespace name1 and I have a using on this particular symbol, variable of name2. So, what happens? If I say v11, what it will check? It will check that I am using namespace name1. So, does v11 exists in that namespace; it does. So, it associates with this. If I say, name1::v12, it will associate here. So, you can see that even when I am doing using it is not mandatory that I will have to use the short form. In place of saying this, I could have

simply written v12; that also would have referred to the same variable in name1, because I have a using namespace name1. But, it is not mandatory I can use the using shortcut or I can also use the fully qualified name, as I have done here. Think about v21.v21 will mean this. Why? I do not have a using on namespace name2. But, I have been using on this particular symbol itself. So, if I say v21 it means that, it is name2 :: v21 and it will be right.

Similarly, I can also directly still refer to this, by saying it is name2::v21; this also is permitted; explicit use of name is permitted. Think of v22; v22 belongs here. I do not have a using on name2 namespace. So, v22 cannot mean this one, this particular entity neither I have a using on, like name2::v22; I do not have that. So, the compiler cannot see any v22 symbol in this code and the compiler will treat this as undefined. It will say, there is no symbol called v22. This a basic use of using; as such, you have been seeing this in the whole code so far I had mentioned at the very beginning that, we will keep on writing this because all of the standard library symbols are in the namespace std. So, writing this makes our life easier otherwise, your cout will have to be written as std :: out; cin as std::cin and so on.

(Refer Slide Time: 21:06)

Module 20

Partha Pratim Das

Objectives & Outline

namespace Fundamental

namespace Scenarios

namespace Features

Nested namespace

using namespace

Global namespace

std namespace

namespaces are Open

namespace vis-a-vis class

Lexical Scope

Summary

Program 20.06: Global namespace

- using or using namespace hides some of the names

```
#include <iostream>
using namespace std;

int data = 0;           // Global Data

namespace name1 {
    int data = 1;     // namespace Data
}

int main() {
    using name1::data;

    cout << data << endl;      // 1 // name1::data -- Hides global data
    cout << name1::data << endl; // 1
    cout << ::data << endl;    // 0 // ::data -- global data
    return 0;
}
```

- Items in Global namespace may be accessed by scope resolution operator

NPTEL MOOCs Programming in C++

Partha Pratim Das

Now, let us also talk about the global namespace. Suppose, I have an example. Let us understand the example; I have a variable written in the global scope data. In the namespace name1, I have a variable data. Now, as such they are resolvable if you do not think about this program given below, they are resolvable, because if I write data, it means this and, if I write name1::data then it means this, clear. But, let us suppose in this function main, I have a using for name1::data, right. So, what does it mean? It means that, now if I talk about data, it means this data; it does not mean this data anymore; because, I have a using on name1::data. So, it says that, name1::data will be known as a data in, from this point onwards. So, if I say data, I get this.

If I say name1::data, I also get this. So, what it means is that, I have lost the ability to get access to the data which was not defined in the namespace, which is defined outside. So, C++ gives us a mechanism to be able to access those symbols, which are in the global space, which are in the global scope. All that you do, you use the same notation, but just that the global, the consideration is kind of that, the global scope also as if at a namespace, but that namespace has no name. So, it is just a blank name. So, all that you do is, just put ::data. So, that will always mean the name in the global scope. So, that is a basic concept of the global namespace that exists.

(Refer Slide Time: 23:15)

The slide is titled "Program 20.07: std Namespace". On the left, there's a sidebar with a logo and a navigation menu listing topics like Module 20, Partha Pratim Das, Objectives & Outline, namespaces Fundamentals, namespaces Scenarios, namespaces Features, Nested namespaces, Global namespaces, std namespaces, Open namespaces vis-a-vis class, Lexical Scope Summary, and NPTEL MOOCs Programming in C++.

The main content area starts with a bullet point: "Entire C++ Standard Library is put in its own namespace, called std".

Without using using std	With using using std
<pre>#include <iostream> int main(){ int num; std::cout << "Enter a value: " ; std::cin >> num; std::cout << "value is: " ; std::cout << num ; return 0; }</pre> <ul style="list-style-type: none"> Here, cout, cin are explicitly qualified by their namespace. So, to write to standard output, we specify std::cout; to read from standard input, we use std::cin It is useful if a few library is to be used; no need to add entire std library to the global namespace 	<pre>#include <iostream> using namespace std; int main(){ int num; cout << "Enter a value: " ; cin >> num; cout << "value is: " ; cout << num ; return 0; }</pre> <ul style="list-style-type: none"> By the statement using namespace std; std namespace is brought into the current namespace, which gives us direct access to the names of the functions and classes defined within the library without having to qualify each one with std:: When several libraries are to be used it is a convenient method

At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" with page number "15".

Standard namespace, so we have been talking about this that, all C++ puts all its standard library symbols, classes, functions, everything in the std namespace. So, if we just include IO stream and want to do, write a program using this, then every time we will have to prefix the symbol with the std:::. For example, if I have to write endl, I have to write it as std::endl, endl. So, we can do a shortcut by putting this using namespace std. What will mean that, if I write cout, it will also check if std namespace has a cout; it does have one. So, std will relate to that. So, std namespace is the most important namespace that is available to us. There are couple of more namespaces also defined. We will talk about those later.

(Refer Slide Time: 24:09)

Module 20
Partha Pratim Das
Objectives & Outline
namespace Fundamental
namespace Scenarios
namespace Features
Nested namespace
Global namespace
std namespace
namespaces are Open
namespace vis-a-vis class
Lexical Scope Summary

Program 20.08: namespaces are Open

- namespaces are open: New Declarations can be added

```
#include <iostream>
using namespace std;

namespace open
{ int x = 30; }

namespace open
{ int y = 40; }

int main()
{
    using namespace open;
    x = y = 20;
    cout << x << " " << y ;
    return 0 ;
}

Output: 20 20
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 16

One very interesting concept about namespace is namespaces are open, in the sense that; think about a class if you just define a class, then whatever symbols you put within that class, the data members, the functions and so on, the friend and all those that has to be put into the one integral definition of the class. And, once that scope is over then you cannot add new symbols to that scope of the class. But in namespace, that is different. So, that is what is meant by the namespaces being open. So here, I have created a namespace open, where I have put a symbol x. The scope is closed; this started here, it closed it here. But then, I again say namespace open and put another symbol; that is... So, what happens is, this will get added to the same scope. Now, it says basically, the

namespace open has two symbols, symbol x as well as symbol y. So, simply if we say using namespace open then I can use both x and y and from the same namespace, they will, x will bind here and y will bind here. So, this openness is an interesting concept which is very flexible, so that, you can, in multiple, different files also, different parts of the namespace may be specified.

(Refer Slide Time: 25:27)

namespace	class
<ul style="list-style-type: none"> • Every namespace is not a class • A namespace can be reopened and more declaration added to it • No instance of a namespace can be created • using-declarations can be used to short-cut namespace qualification • A namespace may be unnamed 	<ul style="list-style-type: none"> • Every class defines a namespace • A class cannot be reopened • A class has multiple instances • No using-like declaration for a class • An unnamed class is not allowed

NPTEL MOOCs Programming in C++ Partha Pratim Das

Since we have been talking, referring frequently to the concept of class and comparing with the namespace this is just a summary of that comparison. Between the namespace and the class every namespace is not a class and every class in turn defines a namespace. It does give you the same qualification ability. Namespaces can be reopened, but and more declarations put into it. In terms of class, there is nothing like that. Certainly, namespaces cannot be instantiated; classes are meant to be instantiated for objects. Using mechanism is available for namespace; certainly, for class there is no sense of doing that. And interestingly, a namespace may be unnamed; I can have a namespace just to segregate some of the symbols and just to put them together, but not to access them from outside. See, why you need the names of the namespace so that from outside, by using the using declaration or directly you can access the symbols inside the namespace. Now, if I just want to blindly hide some symbols and just want them to interact between themselves I can put them in a namespace and not give a name to that namespace. If I do

not give a name to that namespace, nobody from outside that namespace has an access to the symbols within that namespace. So, unnamed namespaces have a meaning; obviously unnamed class is meaningless. It is not allowed.

(Refer Slide Time: 26:56)

The screenshot shows a presentation slide titled "Lexical Scope". The slide content is as follows:

- Module 20
- Partha Pratim Das
- Objectives & Outline
- namespace Fundamental
- namespace Scenarios
- namespace Features
- Nested namespace
- using namespace Global namespace
- std namespace
- Namespaces are Open
- namespace vis-a-vis class
- Lexical Scope
- Summary

Lexical Scope

- The scope of a name binding – an association of a name to an entity, such as a variable – is the part of a computer program where the binding is valid: where the name can be used to refer to the entity
- C++ supports a variety of scopes:
 - Expression Scope – restricted to one expression, mostly used by compiler
 - Block Scope – create local context
 - Function Scope – create local context associated with a function
 - Class Scope – context for data members and member functions
 - Namespace Scope – grouping of symbols for code organization
 - File Scope – limit symbols to a single file
 - Global Scope – outer-most, singleton scope containing the whole program

NPTEL MOOCs Programming in C++ Partha Pratim Das 18

Before I close I would like to just remind you that, namespace belongs to one of the different lexical classes that C++ define. And, this is just to recap that these are the different lexical scopes that you have. The expression scope is what you, where the temporaries are used for computing different parts of an expression and they have a scope within the expression itself. Most often, we do not get to see these temporaries. So, it is only compiler who handles that. But, we have been frequently dealing with the block and function scope, particularly in C; and, also the file and global scope in C. And, having come to C++, these exist and in addition we have the class scope and the namespace scope that we have just discussed.

(Refer Slide Time: 27:51)

The slide has a dark blue header with the title 'Lexical Scope'. On the left is a sidebar with a logo and a navigation menu. The main content area contains a bulleted list about scopes.

Module 20

Partha Pratim Das

Objectives & Outline

namespace Fundamental

namespace Scenarios

namespace Features

Nested namespace

using namespace

Global namespace

std namespace

namespaces are Open

namespace vis-a-vis class

Lexical Scope

Summary

Lexical Scope

- Scopes may be named or Unnamed
 - Named Scope – Option to refer to the scope from outside
 - Class Scope – class name
 - Namespace Scope – namespace name or unnamed
 - Global Scope – “::”
 - Unnamed Scope
 - Expression Scope
 - Block Scope
 - Function Scope
 - File Scope
- Scopes may or may not be nested
 - Scopes that may be nested
 - Block Scope
 - Class Scope
 - Namespace Scope
 - Scopes that cannot be nested
 - Expression Scope
 - Function Scope – may contain **Class Scopes**
 - File Scope – will contain several other scopes
 - Global Scope – will contain several other scopes

NPTEL MOOCs Programming in C++

Partha Pratim Das

19

As you can note that scopes may be named or unnamed; like class scope will always have to have a name. Namespace scope will usually have a name, but may be unnamed also. Global scopes do not have a name, but can be identified by this scope resolution operator. And these scopes, like expression, block, function and file scope, do not have any names they are unnamed. And also, the scopes may be nested, like block scope, class scope, namespace scope; they may be nested. But, there are certain scopes like function scope or expression scope, they cannot be nested. But some of them can contain other scopes, but may not contain themselves.

So, this is just to summarise because it is very important to, specifically know the lexical scope, because C++ happens to be a strongly lexically scoped language. It does not have dynamic scoping; that is, it does not have execution dependent name binding. It is a whole name binding, that is what it associates, how it associates memory with a variable name is completely dependent on the lexical scope, completely dependent on the static time. Therefore, it is very important that out of these all different options of specifying variable names and restricting their visibility and accesses, which is the right one, in a right design situation and use that.

(Refer Slide Time: 29:24)

Module Summary

- Understood namespace as a scoping tool in c++
- Analyzed typical scenarios that namespace helps address
- Studied several features of namespace
- Understood how namespace is placed in respect of different lexical scopes of C++

NPTEL MOOCs Programming in C++ Partha Pratim Das 20

And, namespace only helps to extend that kitty of lexical scopes and particularly, is a powerful tool for organising your code and particularly accessing libraries and segregating your library from symbols of your library from symbols of other third party libraries and so on. For example, if you are developing a library which you want to share out to others, it always a good idea that you put a meaningful namespace name to that whole thing and put that whole development within that namespace, like the standard library is using std and then give it to the user, so that it will not have a possibility that the, you have used certain function names or class names in your library which the user also wants to use, but are not able to do so.

So, that is about the namespaces and we will close here.

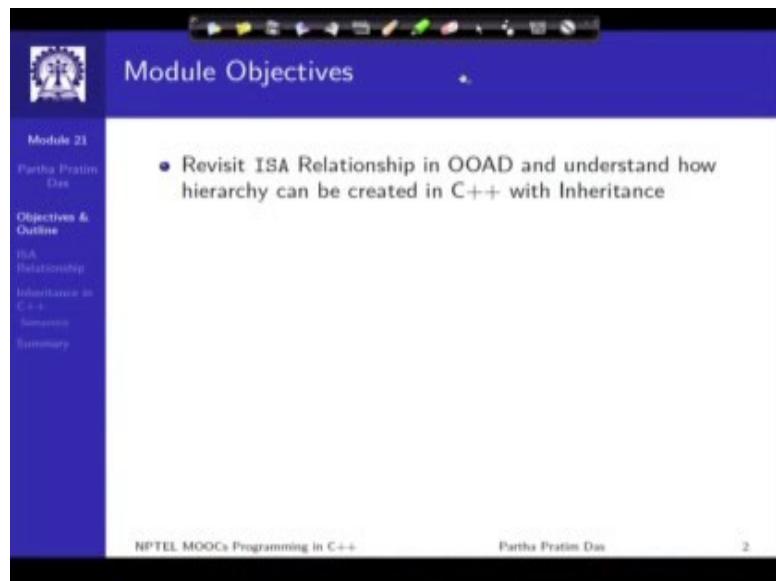
Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 36
Inheritance: Part I

Welcome to Module 21 of Programming in C++. From this module continuing over the next number of modules, we are going to start the discussion on most important aspect of an object oriented programming language that is, the dealing of inheritance amongst classes and amongst objects. We have so far learnt how classes can be defined in C++, how objects can be instantiated. We have learnt about the feasibility of different data members and member functions of a class. We have learnt about the construction and destruction process and different lifetime issues of objects. We have also talked about other features that are related to variety of extension or exceptions to encapsulation and access by functions in terms of friend functions, in terms of static functions and so on, and we have seen how overloading of different member functions and global functions can be done.

Now, inheritance is one topic which kind of will combine all these understanding into building the core backbone of design of object based systems. Therefore, before we start of studying this in depth I would urge all of you to revise and be very thorough about the different features of C++ that we have discussed so far because we would be referring to all of them in a very frequent regularity now together.

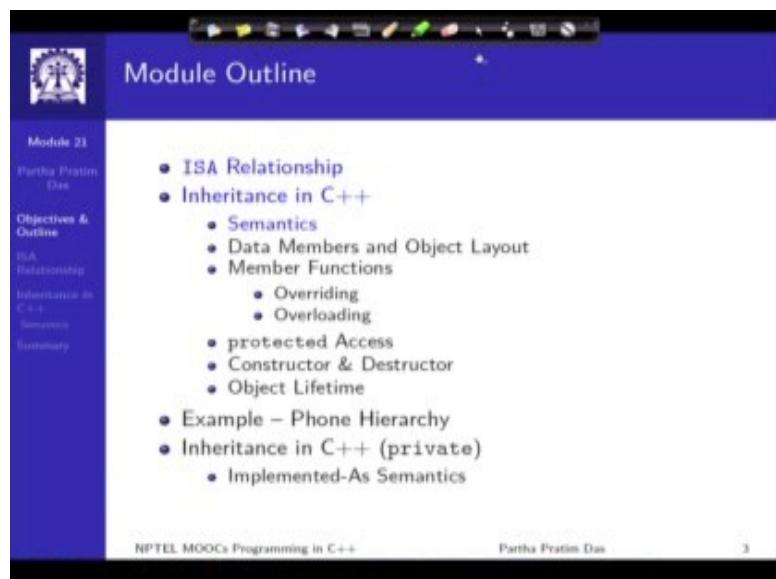
(Refer Slide Time: 02:42)



This slide is titled "Module Objectives". It features a sidebar on the left with the title "Module 23" and the name "Partha Pratim Das". Below this, under "Objectives & Outline", are links to "ISA Relationship", "Inheritance in C++", "Semantics", and "Summary". The main content area contains a single bullet point: "• Revisit ISA Relationship in OOAD and understand how hierarchy can be created in C++ with Inheritance". At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

Now, for discussing inheritance as I have already mentioned this will span over a number of modules. In this particular module, we would try to revisit the ISA Relationship or hierarchy of object oriented analysis and design and see how that can be created in terms of C++ inheritance.

(Refer Slide Time: 03:04)



This slide is titled "Module Outline". It has the same sidebar as the previous slide, showing "Module 23" and "Partha Pratim Das". Under "Objectives & Outline", the "ISA Relationship" link is highlighted. The main content area lists several topics: "• ISA Relationship", "• Inheritance in C++" (which is expanded to show "Semantics", "Data Members and Object Layout", "Member Functions", "Overriding", "Overloading", "protected Access", "Constructor & Destructor", and "Object Lifetime"), "• Example – Phone Hierarchy", "• Inheritance in C++ (private)" (which is expanded to show "Implemented-As Semantics"). At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

The outline that I present here is organized little bit differently. This is not just the outline of this current module. This is the outline of what I intend to discuss at the first level of inheritance in C++. There will be some subsequent sequel to this which we will talk about inheritance in a dynamic scenario, but this is the basic foundational aspects of inheritance. What we will do, as we move from one module to the other I will highlight the specific topics that the particular module we will discuss in terms of blue color. So, if you look into this of this whole context, this is what we intend to discuss in the module 21.

(Refer Slide Time: 04:01)

The screenshot shows a presentation slide with a dark blue header containing the title 'ISA Relationship'. The slide content includes a bulleted list and a small video player in the bottom right corner.

Module 21
Partha Pratim Das
Objectives & Outline
ISA Relationship
Inheritance in C++
Summary
Summary

ISA Relationship

- We often find one object is a *specialization / generalization* of another
- OOAD models this using **ISA** relationship
- C++ models **ISA** relationship by *Inheritance of classes*

NPTEL MOOCs Programming in C++
Partha Pratim Das

A small video player in the bottom right corner shows a man speaking, identified as Partha Pratim Das.

Having said that, let us get started now. So, we are aware of ISA relationship, we have often not talked about it in this course and you may be familiar with this otherwise also, that in the real world we often find that one object is a specialization or generalization of another object. Specialization and generalization these are the key. So, one object is a most specialized or generalized form of the other and this is known as ISA relationship and object oriented analysis and design treats this in depth and C++ model this ISA relationship by inheritance of classes.

(Refer Slide Time: 04:57)

The slide is titled 'ISA Relationship' and is part of 'Module 23'. It features a sidebar with navigation links: Partha Pratim Das, Objectives & Outcome, ISA Relationship, Inheritance in C++, Summary, and Summary. The main content area contains four bullet points with UML class diagrams:

- Rose ISA Flower**
 - Rose has the properties of Flower – like fragrance, having petals etc.
 - Rose has some additional properties – like rosy fragrance
 - Rose is a specialization of Flower
 - Flower is a generalization of Rose
- Red Rose ISA Rose**
 - Red Rose has the properties of Rose – like rosy fragrance etc.
 - Red Rose has some additional properties – like it is red
 - Red Rose is a specialization of Rose
 - Rose is a generalization of Red Rose

```
graph LR; RedRose[Red Rose] --> Rose[Rose]; Rose --> Flower(Flower)
```
- TwoWheeler ISA Vehicle; ThreeWheeler ISA Vehicle**

```
graph LR; TwoWheeler[TwoWheeler] --> Vehicle[Vehicle]; ThreeWheeler[ThreeWheeler] --> Vehicle;
```
- Manager ISA Employee**

```
graph LR; Manager[Manager] --> Employee[Employee]
```

On the right side of the slide, there is a circular portrait of Partha Pratim Das.

So, before getting into the inheritance logic of C++, let us quickly take a look into the ISA relationship. Let say what do you say is a specialization generalization? Suppose, we say that rose ISA flower. So, what do you mean by saying rose ISA flower. We say mean that rose, this is the special side, this is the general side that rose is a specialization of flower that is it has all the properties that flower has, like whenever we say flower; certain concepts come to our mind, like it will have fragrance, it will have bright color, it will have petals and so on. So, rose has all the properties that a flower is suppose to have. But in addition, rose has some additional properties that is why you would like to specialize that. For example, a rose has very specific rosy fragrance which is not similar to the fragrance of many of the other flowers.

So, when this is done we say that rose is a specialization of flower and we can say the same thing in a reverse way. We can say that flower is a generalization of rose. So, if we have again a red rose and a rose, we can say a red rose ISA rose in the same way red rose has all the properties of a rose, but it has an additional property like its color is red and this is the generalization specialization that will exist between rose and red rose. So, we designate that in terms of what I have drawn here, this is known as many of you may know this is known as UML diagrams. Unified Modeling Language is a strong language to describe systems in the object oriented way. So, these designate the classes and this

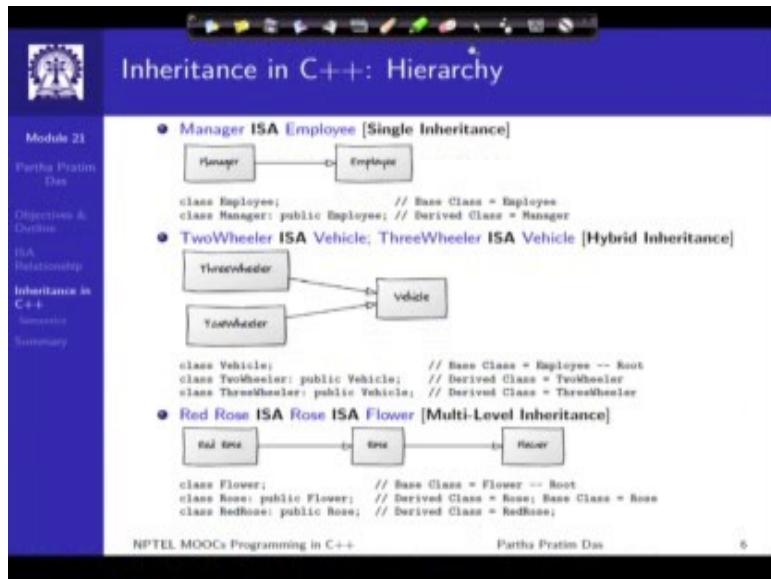
arrow which ends with open triangle at the end means specialization.

So, this is to be read; rose ISA flower that is the direction of the arrow, red rose ISA rose and so on. So, diagrammatically we can we will often depict the inheritance or specialization generalization in terms of this diagrams. Similarly, we can have; we can say that Twowheeler ISA vehicle, a vehicle which we use to move around. So, we can move around in Twowheelers, so we can say that Twowheeler ISA vehicle. Similarly, Threewheelers ISA vehicle, but we can designate that in terms of vehicle being the generalization and 3 wheelers and 2 wheelers are two different specialization of the vehicle.

So, the generalization and specialization could relate in multiple different ways. So, this is a one kind where there is one generalized concept and there are multiple specialized concepts for that. We usually refer to this as a base and these, we refer to as derived. Different languages will use different nomenclature, for example, those of you who are familiar with Java, we will identify this base or the generalization as super class and derived or specialization as subclass. In some cases, the particular class which is the most generalized is often referred to as a root.

Yet another example of ISA relationship could be from the employee domain. We say manager ISA employee which means that manager can perform all functions of an employee, but an employee can perform some; manager can perform some functions that the employee cannot do, possibly the manager can hire employees which employees themselves cannot do. So, this is the basic concept of ISA relationship which we want to bring in in the context of C++ programming.

(Refer Slide Time: 09:20)



So, in this, let me illustrate that if I want to encode this information of generalization specialization in C++, how do we go about doing that? So, the simplest of these cases is the manager ISA employee, where we have just two classes one is the specialization of the other. So, we normally say refer to this as a single inheritance because you have just one relationship here. So, we write a class, say the representation for employees, class employee. Here, I have just used a kind of you know incomplete definition of a class. This does not have the definition, this is just saying that there is a class called employee because we will look into the data member and member function details later on.

We are just interested to define the relationship between classes. So, given that, now we say that the manager is an employee through this form. So, here you can see that the critical addition that are making is after the class manager, there is a separator : and then I am putting a new word public and then the name of the employee class and this is to be read as in the OOAD terms, this will be read as manager ISA employee or manager derives from the employee. So, employee is the base class and the manager is a derived class.

In the two wheeler-three wheeler vehicle example; given this diagram we are actually dealing with hybrid inheritance. Hybrid inheritance is a context where one base class has

more than one specialization and we can easily encode that in C++ in terms of a root class which is the vehicle. Then we say Twowheelers is a vehicle by saying Twowheeler : public Vehicle. Threewheelers is a vehicle by Twowheelers :public Vehicle. So, there is one root class and there one base class and there are two derived classes in this.

Finally, coming to the rose example, we have what is known as a multilevel inheritance because this is one level, this is another level. So, together it is a multilevel inheritance RedRose ISA Rose ISA flower. So, here flower is the base class and the derived class is Rose , but if we look into the next pair of relationship then Rose turns out to be the base class and RedRose turns away to be the derived class. So, whether a particular class is a base class or a derived class cannot be decided absolutely just based on that class, it depends on where does the class lies on the hierarchy.

The classes which are leaf, at the leaf that is, which is not you know specialized by any other class are necessarily derived class. A class which does not have any super class or parent is necessarily the base class and the root class, but classes like who exist in the middle could be derived class for one part of the inheritance and base class for the other part of the inheritance, and we could really have variety of different complex and hybrid kind of inheritance structure that we would like to encode and understand in C++.

(Refer Slide Time: 13:02)

The slide is titled "Inheritance in C++: Phones". It features a sidebar with course navigation: Module 23, Partha Pratim Das, Objectives & Outline, ISA Relationship, Inheritance in C++, Resources, and Summary. The main content area is divided into three sections:

- Landline Phone**
 - Call: By dial / keyboard
 - Answer
- Mobile Phone**
 - Call: By keyboard – shows number
 - By Number
 - By Name
 - Answer
 - Redial
 - Set Ring Tone
 - Add Contact
 - Number
 - Name
- Smart Phone**
 - Call: By touchscreen – shows number & photo
 - By Number
 - By Name
 - Answer
 - Redial
 - Set Ring Tone
 - Add Contact
 - Number
 - Name
 - Photo

A callout box highlights:

- There exists a substantial overlap between the functionality of the phones
- A mobile phone is more capable than a land line phone and can perform (almost) all its functions
- A smart phone is more capable than a mobile phone and can perform (almost) all its functions

*** These phones belong to a Specialization / Generalization hierarchy**

At the bottom, it says NPTEL MOOCs Programming in C++ and Partha Pratim Das.

So, here I just take to you a little different kind of an example. We all are familiar with phones, you all use variety of phones. So, I am saying that, if we consider just 3 common types of phones that we use today; the landline phone, the mobile phone and the smart phone, then we can see that there is; and all that I am trying to do here is kind of associate the functionality that we mentally associate with these phones. So, say landline phone, what will be the functionality that you will associate; that you can make a call, you can answer a call.

For a mobile phone you can obviously, do all of that so, but you can do so many other things, possibly in a mobile phone you will be able to redial the last number, you will be able to set a ring tone, you will possibly have an address book where you can put your contacts by number and name and so on so forth, there would be several others. So, I can see that the basic functionality of a landline phone is also satisfied by the mobile phone, this already exist. So, I can say that a mobile phone ISA landline phone; this ISA does not mean that it is same as, but this means that the mobile phone can satisfy all the functionality that the landline phone can satisfy.

Similarly, if I come to a smart phone, then I will have again these functionalities of call and answer, redial all this, but I may have some additional functionality. Typically that I

can associate a photograph with my contact and if I associate a photograph with my contact then possibly smart phone at the time of call or at the time of answering, receiving the call, at the time of redialing will be able to show that photograph. So, there is commonality between; there is a substantial commonality between these different kinds of phone, but as we go from one kind of phone to the other, landline to mobile we find that there are some additional functionality that come in. As we go from mobile to smart phone we find yet some more functionalities coming in and in that way the phones form a very nice small specialization generalization hierarchy.

So, this was just to sensitize you, just to make you aware of the different kinds of gadgets and their hierarchies that are possible. At a later module we would like to take up the phones and actually do a complete design of the inheritance structure but now, let me move on to the concrete semantics of what the C++ inheritance would mean.

(Refer Slide Time: 15:56)

The screenshot shows a presentation slide with a blue header bar containing the title 'Inheritance in C++: Semantics'. On the left, a vertical sidebar lists navigation links: 'Module 23', 'Partha Pratim Das', 'Objectives & Outline', 'ISA Relationship', 'Inheritance in C++ Semantics Summary'. The main content area contains the following text and diagram:

- **Derived ISA Base**

Diagram illustrating ISA relationship:

```
graph LR; Base --> Derived;
```

Code example:

```
class Base; // Base Class = Base
class Derived: public Base; // Derived Class = Derived
```

- Use keyword **public** after class name to denote inheritance
- Name of the Base class follow the keyword

At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

So, in general we will talk about two classes; the base class and the derived class. So, the ISA model is; Derived ISA Base. So, the name of the derived class is derived and the name of the base class is base and certainly we have already seen that this is how this is to be represented in C++. So, the new introduction is this public keyword as a keyword it already exist because we have used it for access specification, but here we are using it

again for some special purpose of inheritance, and we will see what is the significance of this particular keyword and we will talk about their alternates, but for now just take it kind of as a prescription that if I want to say that the Derived ISA Base then this is a way to say that, and after this keyword you can put the class name which is the generalized or the base class name that you need to specify.

Now, having said that certainly tokens up a whole lot of a question because as we will expect that the base will have different data members. The derived will also have different data members, base will have different methods, derived will also have different methods. Base will need to be constructed, derived will need to be constructed and so on.

(Refer Slide Time: 17:49)

The slide is titled "Inheritance in C++: Semantics". The sidebar on the left lists topics: Module 23, Partha Pratim Das, Objectives & Outline, ISA Relationship, Inheritance in C++, Semantics Summary. The main content area contains the following bullet points:

- Derived ISA Base
- Data Members
 - Derived class *inherits* all data members of Base class
 - Derived class may *add* data members of its own
- Member Functions
 - Derived class *inherits* all member functions of Base class
 - Derived class may *override* a member function of Base class by *redefining* it with the *same signature*
 - Derived class may *overload* a member function of Base class by *redefining* it with the *same name*, but *different signature*
- Access Specification
 - Derived class *cannot access private* members of Base class
 - Derived class *can access protected* members of Base class
- Construction-Destruction
 - A *constructor* of the Derived class *must first* call a *constructor* of the Base class to construct the Base class instance of the Derived class
 - The *destructor* of the Derived class *must* call the *destructor* of the Base class to destruct the Base class instance of the Derived class

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, what we need to very carefully specify and understand is; what is the semantics of, how this base derived relationship work? So, I would just like to outline at the very beginning and then we will take up each one of these and try to do a more detailed discussion. I would first like to outline that when you talk about semantics of inheritance in C++, these are the first level items that you need to be very careful about. The first thing is, I am sorry; the first thing is data members. So, we will say that the derived class inherits all data members of the base class. So, even if you do not have any data member in the derived class, it will still have a lot of data members which are all the data

members of the base class, but it can further add more data members.

I will come to examples later on for each one of these, but I am trying to tell you the basic principle. The basic principle of inheritance is I have some concepts which is this and I am trying to specialize to provide a more special concept which need to satisfy everything that the general concept satisfies. So, the base, the derived will have to satisfy everything that the base satisfies. So, it needs to have all the data members, but it can add its own to further refine the whole concept.

In terms of member functions again, we will see that there are very key ideas that come in. The first part is the same that a derived class inherits all the member functions of the base class, but then there is a major difference that can; as you inherit a member function you have the choice of re-implementing it redefining it without actually changing its signature. That is you want to again define the same function with a different kind of algorithm and if you do that then you say that you are overriding the member function in the base class; be very cautious because this concept sounds extremely close to the concept of overloading and therefore, there is a good possibility that you will start confusing it with the overloading and what makes things worst is you can actually also have overloading in the context of inheritance.

The difference is when you redefine a function with the same signature it is overriding. When you redefine a function with the different signature it is called overloading. So, these are the different semantics that we will need to understand in terms of the member functions behavior in under inheritance.

The next aspect that we need to look into is in terms of access specification. Now, naturally in terms of access specification we know that there is private access and there is public access. Private access is for the members of the class, public access is for everybody. So, a derived class of a base class is certainly not a part of that base class. So, a derived class cannot access the private members of the base class. It can access only the public members, but then we will see that that will lead to a lot of difficulties because the semantics of specialization that we are trying to put together particularly with reference to overriding will become extremely difficult to code, if the derived class do

not have any access to the internals of the base class. So, with that a new type of access specifier, the protected access specifier is created to support a good semantics of inheritance. So, we will have to learn and understand about, what is this third type of visibility or access specification that is supported in inheritance.

Finally, naturally objects need to be constructed/destructed and when we have inheritance between the base and the derived class, then a derived class instance will need a derived class constructor to be called, but the derived class constructor in turn will have to construct a base class object by calling the base class constructor. So, this is the basic understanding that we will need to build up as to how the base class object and the derived class object get related between themselves and how they can interact.

Similarly, when a derived class object will need to be destructed then the destructor of the base class will have to be invoked, so that you can destroy the base class part of the derived class object. So, this is just the tip of all the major semantics that we need to understand in terms of understanding how to use inheritance and how to model different real world scenario of hierarchy in a very efficient manner in the C++ language.

(Refer Slide Time: 23:53)

So to summarize, we have in this just revisited the hierarchy of OOAD hierarchy concept

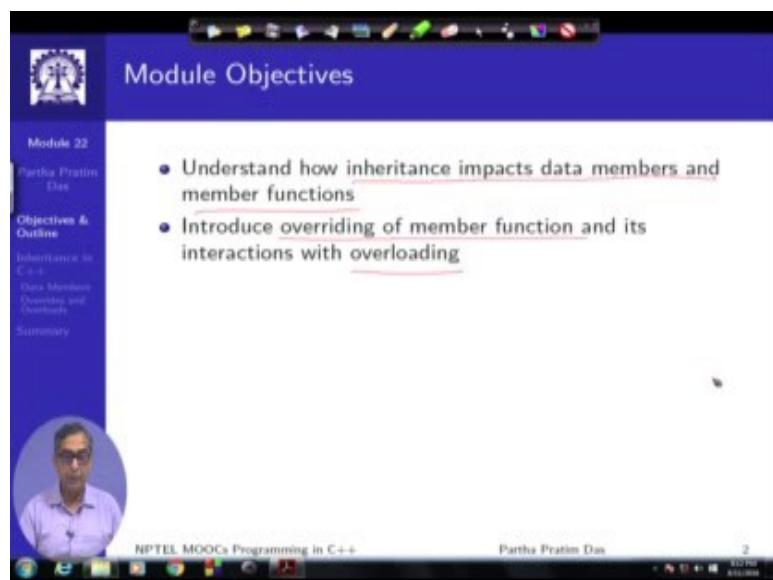
of ISA relationship and class hierarchy concept of object oriented analysis and design, and we have introduced the basic notion of inheritance in C++ and noted what are the different aspects of semantics that we need to understand, that we need to really be master of so that we can use inheritance in an effective manner.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 37
Inheritance: Part II

Welcome to Module 22 of programming in C++. Since the last module, we have been discussing on inheritance; we have noted that inheritance is a basic mechanism in C++ of modelling, encoding, there ISA relationship of object oriented programming, the generalization and specialization relationship.

(Refer Slide Time: 00:33)



The screenshot shows a presentation slide titled "Module Objectives" for "Inheritance in C++". The slide contains the following text:

- Understand how inheritance impacts data members and member functions
- Introduce overriding of member function and its interactions with overloading

The slide has a blue header and footer. The footer includes the NPTEL logo and the text "NPTEL MOOCs Programming in C++". There is a video player interface at the bottom with a thumbnail of the speaker and playback controls.

And, we have just seen how basic inheritance information can be coded in the language. In this module, we would try to discuss how inheritance impacts data members and member function that is under inheritance, what happens to the data members and member functions. And we will try to understand that what is overriding of member functions, and how does it interact with the concept of overloading that you are already familiar with.

(Refer Slide Time: 01:24)

This slide shows the module outline for Inheritance in C++. The title is "Module Outline". On the left, there is a sidebar with the following navigation links: Module 22, Partha Pratim Das, Objectives & Outline, Inheritance in C++, Data Members, Overriding and Overloads, and Summary. Below these links is a circular video player showing a man speaking. The main content area contains the following outline:

- ISA Relationship
- Inheritance in C++
 - Semantics
 - Data Members and Object Layout
 - Member Functions
 - Overriding
 - Overloading
 - protected Access
 - Constructor & Destructor
 - Object Lifetime
- Example – Phone Hierarchy
- Inheritance in C++ (private)
 - Implemented-As Semantics

At the bottom of the slide, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das". There is also a small number "3" in the bottom right corner.

This is the outline and as I had mentioned in the last module, this is the total outline of inheritance; and the blue part is what we will be discussing in this current module.

(Refer Slide Time: 01:41)

This slide is titled "Inheritance in C++: Semantics". It has the same sidebar as the previous slide. The main content area contains the following outline:

- Derived ISA Base
- Data Members
 - Derived class *inherits* all data members of **Base** class
 - Derived class may *add* data members of its own
- Member Functions
 - Derived class *inherits* all member functions of **Base** class
 - Derived class may *override* a member function of **Base** class by *redefining* it with the *same signature*
 - Derived class may *overload* a member function of **Base** class by *redefining* it with the *same name*, but *different signature*
- Access Specification
 - Derived class *cannot access private* members of **Base** class
 - Derived class *can access protected* members of **Base** class
- Construction-Destruction
 - A *constructor* of the Derived class *must first* call a *constructor* of the **Base** class to construct the **Base** class instance of the Derived class
 - The *destructor* of the Derived class *must* call the *destructor* of the **Base** class to destruct the **Base** class instance of the Derived class

At the bottom of the slide, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das". There is also a small number "4" in the bottom right corner.

So, to recap, this is the inheritance aspects of inheritance semantics. So, we will need to understand about what happens to the data members, member function, access

specification, construction, destruction, object lifetime, all these aspects. And here we are discussing starting to discuss about the data members.

(Refer Slide Time: 02:04)

Inheritance in C++:
Data Members and Object Layout

- Derived ISA Base
- Data Members
 - Derived class *inherits* all data members of Base class
 - Derived class may *add* data members of its own
- Object Layout
 - Derived class *layout* contains an instance of the Base class
 - Further, Derived class *layout* will have data members of its own
 - C++ does not guarantee the *relative position* of the Base class instance and Derived class members

Now if derived ISA base, ISA basic is a relationship that we are trying to encode. Then we will observe that the derive class will inherit all data members of the base class. This is true in every case of inheritance. Further, the derive class may add more data members of its own it can have some more additional data members. So, if we have that then certainly in terms of the layout of the object, you would recall that we had talked about layout of objects in couple of earlier modules. And if you are not clear about the concepts, I would add that you go back, and revisit those modules before you proceed further. So, the layout is basically the organisation of the different data members within an object of a class in the way that they will get organised in memory.

The important point to be noted is that derived class, the layout of the derive class will contain an instances of the base class. This is something which is new that will come in here. And further naturally, the derive class can have its own members. And unlike the case of sample data members, the relative position of the base class instance, and the derive class members are not guaranteed in C++.

(Refer Slide Time: 04:00)

The slide title is "Inheritance in C++: Data Members and Object Layout". It features a UML class diagram showing Class B (base) with data members data1B_ (private) and data2B_ (public). Class D (derived) inherits from B and adds infoD_. Below the diagram, two objects are shown: Object b (a circle containing data1B_ and data2B_) and Object d (a rectangle containing data1B_, data2B_, and infoD_). A note states: "d cannot access data1B_, even though data1B_ is a part of B! d can access data2B_." The slide also includes code snippets for classes B and D, and a navigation bar for NPTEL MOOCs Programming in C++.

Let us, go to an example and start seeing what we mean. So, we start with a base class definition, B is a base class; it has two data members, I have just arbitrary taken two data members, I am not showing the member functions yet, just showing the data members, so data1B_, which as I have put is a private member and data2B_, which is a public member. Then we have another class D, which is public B; that is it is a specialization. So, if I draw in terms of the UML notation, then this is what we are representing here, D ISA B. Now what happens is when we look try to look for the data members of D certainly, what we are saying is guaranty that we will have a data members data1B_ in the class D as well even though we will not be writing this explicitly.

Similarly, we will have a data member data2B_ in class D by inheritance from class B. So, these are therefore, this is why I am highlighting here these are the data members, which are inherited; they do not have to be there cannot be listed in this class. But the fact that D ISA B will inherit this in every instance of D. But I can have some additional members also like the infoD_ that I am putting in here, which is a member of D and was not a member of B; so, this will exist only in this.

In this context, if I define if I instantiate an object B, then I would certainly have a layout for this, which would look like this is what we have discussed already. And in terms of a

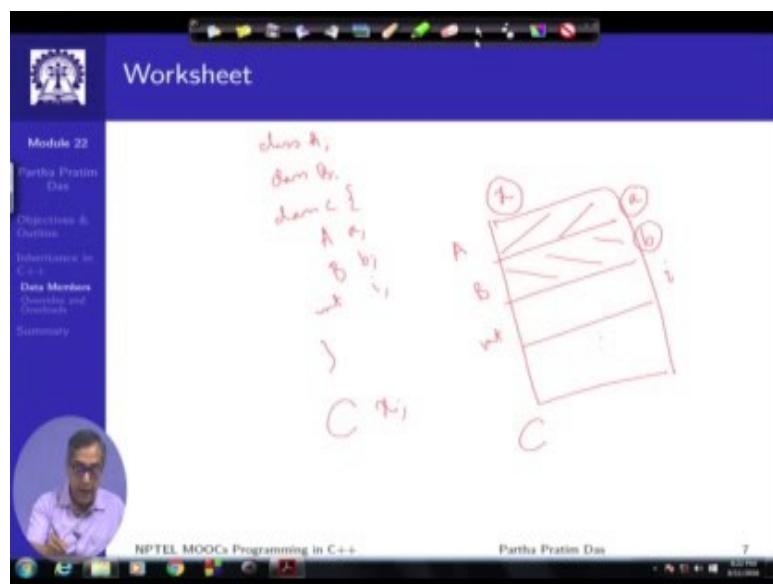
normal layout what we have discussed that if a class is instantiated, then this is the object and what is the basic properties of the layout. There are two basic properties that we had discussed one is all data members will be allocated space in the object in the memory in a contiguous manner, that is the whole of a object as a starting address and a size which is a number of bytes of memory that it occupies.

And all of the data members will occur within this space summer. It will never get fragmented that one part of the object is in one part of memory and other part is another part of the memory. So, contiguity is a basic requirement of layout that is we have already seen these, that is one. The second that we had seen is the order that is if we have two data members like this then one will come after. So, they will occur in the order in which they have been put down in the definition of the class this order will be maintained.

This is the basic structure of a layout. So, he given that if I look into the layout of B, this value except something like this. Now let us try to consider, what is the layout of D. Now, in D what happens specifically is certainly D will have its own data member which is which is fine easy to understand. But since D ISA B it will have an explicit instance of a B object as a part of the D object. So, as if the D object here, would have in one part a B object and when I am saying that D inherits this data member from B it means; that actually if I talk about D.data1b_, then this particular memory will be referred to. And what is this memory? This memory is a base object part of the derived object.

Similarly, if I refer to D.data2B_ then I am referring this particular part of the memory, which has been inherited through the mechanism of inheritance and actually form the base part of the object.

(Refer Slide Time: 09:46)



So, you can see that there is something very interesting happening here, we had seen that earlier also we had seen; that lets say if I just take an example of different kinds of data members. Let us say if I just go here and let us let me take that I have a class A, I have a class B, I have a class C, which as an A object as a data member, which as a B object as a data member possibly and int and so on and so forth.

Then when I look into the instance of C, say x is an instance of C when I look at x, then I will have a object A, will have a b object B, will have an int i and so on and so forth. So, we have already seen that different objects of different classes could be part of the layout of another class. So, this is the class C and object x of class C, which have instances of different other class and that is the concept of component.

The slide title is "Inheritance in C++: Member Functions – Overrides and Overloads". It features a sidebar with "Module 22" and "Partha Pratim Das" along with navigation links for objectives, outline, inheritance, data members, overrides, overloads, and summary. A video player shows a man speaking. The main content lists bullet points about derived classes:

- Derived **ISA** Base
- Member Functions
 - Derived class *inherits* all member functions of Base class
 - Derived class may *override* a member function of Base class by *redefining* it with the *same signature*
 - Derived class may *overload* a member function of Base class by *redefining* it with the *same name*, but *different signature*
 - Derived class *may add* new member functions
- Static Member Functions
 - Derived class *does not inherit* the static member functions of Base class
- Friend Functions
 - Derived class *does not inherit* the friend functions of Base class

At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

But here as we note that I am sorry, here as we note that just one second, is change our create problems at times.

(Refer Slide Time: 11:06)

The slide title is "Inheritance in C++: Data Members and Object Layout". It has the same sidebar as the previous slide. The main content shows code for classes B and D:

```
class B { // Base Class
    int dataB_1;
public:
    int dataB_2;
    // ...
};

class D : public B { // Derived Class
    // Inherits B::dataB_
    // Inherits B::dataB_
    int infoD_; // Adds D::infoD_
public:
    // ...
};

B b;
D d;
```

Below the code is a diagram titled "Object Layout" showing two objects, "Object b" and "Object d". "Object b" contains "dataB_1" and "dataB_2". "Object d" contains "dataB_1", "dataB_2", and "infoD_". A handwritten note in red says "d has b" with arrows pointing from "Object d" to "Object b". Another note says "d cannot access dataB_1 even though dataB_1 is a part of it!" with an arrow pointing to "Object d". At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

Here, as we note that since D ISA B it also gets a component, but that component is very unique like I can have any number of data members, I may not have any data members. But when I inherit I will have one unique component coming in the derived class object,

which is the base component and that is something that you will have to be very aware of in terms of the object layout.

There some additional noting have made here, which is a immediately not very important, but you can note that since this is a private member of the base class, private member cannot be access from outside. So, I cannot, though I can write `d.data1B_` and that refers to this memory I cannot actually access it, that access is not because is not a part of it, but the access is restricted. Whereas, if I talk about this other member which is public, I can I writing `d.data2B_`, I can also access, that we will discuss about this access issues later on, but I just wanted to highlight this on the passing.

(Refer Slide Time: 12:25)

The screenshot shows a presentation slide titled "Inheritance in C++: Member Functions – Overrides and Overloads". The slide content is organized into sections and bullet points:

- Module 22
- Partha Pratim Das
- Objectives & Outline
- Inheritance in C++
- Data Members
- Overrides and Overloads
- Summary

Inheritance in C++:

- Derived **IS A** Base
- Member Functions
 - Derived class *inherits* all member functions of Base class
 - Derived class may *override* a member function of Base class by *redefining* it with the *same signature*
 - Derived class may *overload* a member function of Base class by *redefining* it with the *same name*; but *different signature*
 - Derived class *may add* new member functions
- Static Member Functions
 - Derived class *does not inherit* the static member functions of Base class
- Friend Functions
 - Derived class *does not inherit* the friend functions of Base class

NPTEL MOOCs Programming in C++ Partha Pratim Das 8

Now, let me move on to checking what happens with the member function with the member functions again it inherits all the member functions, but the important thing is after inheriting it can override a member function or it can overload a member function. So, we need to understand these concepts very, very carefully.

So, I illustrate that with example, but in the passing I would just like you to note, that the static member function or the friend member function cannot be inherited by the inheritance mechanism or in very simple terms. It is only the member functions, non-

static member functions or only those functions, which have this pointer or which is actually associate with an object instance can only be inherited through this mechanism. So, we will see that for the static and friend function the meaning of inheritance would be quite different.

(Refer Slide Time: 13:38)

The slide has a blue header with the title 'Inheritance in C++: Member Functions – Overrides and Overloads'. On the left, there's a sidebar with 'Module 22' and 'Partha Pratim Das'. Below that is a 'Table of Contents' with 'Objectives & Outline', 'Inheritance in C++', 'Data Members', 'Overrides and Overloads', and 'Summary'. A circular video thumbnail of the speaker is also present.

Inheritance	Override & Overload
<pre>class B { // Base Class public: void f(int i); void g(int i); }; class D: public B { // Derived Class public: // Inherit B::f(int) // Inherit B::g(int) }; B b; D d; b.f(1); // Calls B::f(int) b.g(2); // Calls B::g(int) d.f(3); // Calls B::f(int) d.g(4); // Calls B::g(int)</pre>	<pre>class B { // Base Class public: void f(int); void g(int i); }; class D: public B { // Derived Class public: // Inherit B::f(int) void f(int); // Overrides B::f(int) void f(string); // Overloads B::f(int) // Inherit B::g(int) void h(int i); // Adds D::h(int); }; B b; D d; b.f(1); // Calls B::f(int) b.g(2); // Calls B::g(int) d.f(3); // Calls B::f(int) d.g(4); // Calls B::g(int) d.f("read"); // Calls D::f(string) d.h(5); // Calls D::h(int)</pre>

● D::f(int) overrides B::f(int)
● D::f(string) overloads B::f(int)

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, let us go to an example and try to understand what the member function inheritance mean. I would first want you to look at the left part of the code only do not look at this part, do not look at this part, this part as if it is not there. So, I have a class B, I have a class D the base and the derived the derived relationship is the same and for simplicity again in this case I am not looking at the data members I am just looking at the member functions where in public.

Certainly the issue of access is not a problem. Now I have a function f and a function g in class B, but I do not have any function in class D, class D is just an empty class. But still it will inherit the function f, it will inherit the function g and clarify one more notation here is this notation which we have mentioned earlier that a class is a namespace. So, if I am outside of this class then the name of this function is B :: f. So, by B :: f (int) I basically mean that this base class function, by this I mean this base class.

So, what that I am saying is if you inherit then you will automatically have this base class functions available to you.

So, what does that mean, here again do not look at this, what does it mean that, if I have two instances b and d, and I try to invoke this functions f and g. Naturally if I do this is a straight forward we already understand that b.f will invoke. This function b.g will invoke this function there is no surprise in that.

But the interesting thing is I can actually invoke d.f, which is not there in the definition of the class D of which small d is an object is an instance, but even then I can invoke d.f, because d inherits from b and b has f function. So, when I invoke d.f then I am actually invoking the f function of the base class. Similarly, if I do d.g I am invoking the g function of the base class this is what is the core and flux of inheritance of member functions that you simply inherit all the member functions from your parent.

Now, let us forget about this part and look into the right side. Again, the same base class there is no difference I have not changed it. So, in the derive class it inherits f it inherits g as we have seen. In terms of g we again also I have not done anything, but in terms of f I have done something interesting; what I have done? That the (Refer Time: 16:28) signature or the prototype of f that was included in b as a member function have put that prototype again in d.

Now when I do this I say that I am overriding the earlier function, because the moment I put this I am talking about two functions, this function is b : f and this function is d : f. When I did not have this function and still wanted to use it I was only meaning the base class function, but now I explicitly have another definition of the same member function by the same signature. So, let me clear up again. So, if I look in here again to instances of objects if I call f and g with b object certainly the effect is the same. If I call f with the d object then what will happen, earlier d did not have an f, but now it has overridden this f. So, if I now called d.f it no more means this function it does not mean this it means this function, so it has changed into d : f.

So, if you just compare with the previous one d.f (3) was the f function of b now it is f function of d and this behaviour of member functions under inheritance is known as overriding. In overriding, whenever you talk of overriding it means that you are trying to attach two different definitions to a function by the same name and necessarily the same signature. You remember that for overloading the necessary part is the signature as to be different, signature in the sense of at least one parameter as to be different between two overloaded functions the name function name is same the return type does not matter, but the function parameters must differ some other signature must be different.

For overriding, it is that signature must be same, but they must belong to two different classes, which are related by a generalization, specialization, inheritance, relationship. And then depending on which particular object you are using you will end up calling different versions of this function. That is the reason that when we when you b.f you call one function you call this function, but when you do d.f now you call a different function the overridden function. This is the basic mechanism of overriding member functions, which add a lot of new value and semantics to the whole story.

Now at the same time if you look at the other function g in b that also is inherited by d, but that function as not been overridden, that is in d we have not included a fresh signature fresh signature of function g. So, if I in this context if I call d.g then we will again actually be calling the function in b, because that is a function that is inherited. So, this behaviour d.g(4) between the left and the right does not change, because g has not been inherited it has not been overridden after inheritance in the class D. So, this clearly explains as to what is the difference between inheriting impure inheritance and inheritance and overriding of a function. What adds a very nice flavour to the whole thing is I could also overload as function, which the base class has.

Now look into this two and this is this often confuses students and you know. In terms of what I am doing, am I overriding am, I overloading what am I doing. Here the distinguishing features are that the signature of the function is different between these two cases. So, you would do not perceive that it overriding you perceive as if the class D is introducing a new function just incidentally, which as the same name as a function that d already inherits from b.

So, if I invoke something like this d.f again, but with the parameter which is a string C string, which can be taken as a string type in the stl as well. Then between this d.f and between this d.f that is between these two functions, the normal resolution of overloading will work. And depending on the type of the parameter this associates with this function and this associates with this function and this function is necessarily the overload of the other function f, which we inherited and subsequently override.

So, overloading in contrast to overriding would necessarily require that one both functions that are overloaded must belong to the same class, either directly or by inheritance and to they must differ in the signature that they have that is at least to one of the parameters have to be different between these two functions. And that is when we will say that we have a situation of overloading that is taking place.

Beyond all this certainly the class can introduce new functions, which means that a function with a new name, which has no name commonality with the member functions that the base class had neither f nor neither f nor g is an another function h, which the class D adds. And therefore, the class D can actually invoke that function on an object instance and certainly this function will get added.

As we see that in terms of member functions there are four possible dynamics that can happen, one is you could just inherit something as in g you could inherit and override something as in this f that is signature is same. But you are redefining, so possible in the implementation of the function that is this function will be implemented as b.f (int) when you will add body to this. And the implementation of this function which will be D :: f add body to it these will possibly be different so, but signature is the same we have the second case which is overriding, the third is between two member functions you can continue to have overloading as we had earlier. In the same scope two functions by the same name must have different set of parameters to be overloaded and the fourth is we can still add new member function.

So, the class and expand the functionality of the class. And these principles are kind of if my may so are recursive in nature. So, that d could also now become a base class for some another class and if d is a base class for another class, then all of this functions this

overridden f, the overloaded f. The invisible g, which has been, which has come here through inheritance and h, all this four functions would be available to be inherited by any class that specialises from the class D.

(Refer Slide Time: 26:40)

The slide is titled "Module Summary" and is part of "Module 22" by Partha Pratim Das. It includes a sidebar with course objectives and a summary of inheritance in C++. A video thumbnail of the instructor is present, along with the NPTEL logo and copyright information.

- Discussed the effect of inheritance on Data Members and Object Layout
- Discussed the effect of inheritance on Member Functions with special reference to Overriding and Overloading

To summarize here, in this module continuing on the discussion on inheritance, we have discussed two very core concepts of inheritance relating to what happens to the data members when one class specialises from another, we have seen that. Under that situation, the derive class object will have an instance of the base class object as a part of it. And we have noted that in terms of the layout. It is not guaranteed as to, whether the base class object will come at the lower address and the data members of the derived class will occur at a higher address or vice versa or some other mix will be done.

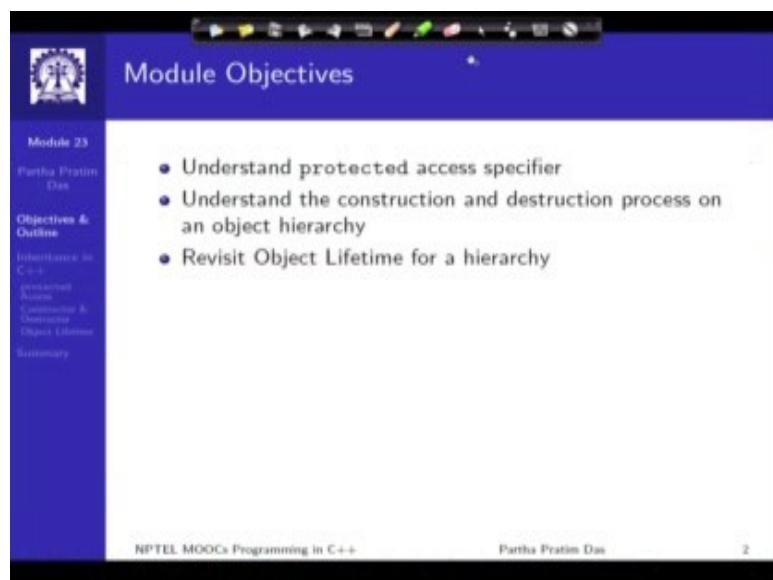
And we have also seen that in terms of inheritance, the member functions are inherited from a base class to the derived class, but very interestingly after inheritance the member functions can be overridden. And in that context, the original rules of overloading also would continue to work.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 38
Inheritance: Part III

Welcome to Module 23 of Programming in C++. We are discussing about Inheritance in C++.

(Refer Slide Time: 00:26)



Module Objectives

- Understand protected access specifier
- Understand the construction and destruction process on an object hierarchy
- Revisit Object Lifetime for a hierarchy

Module 23
Partha Pratim Das
Objectives & Outline
Inheritance in C++
protected access
Construction & Destruction
Object Lifetime
Summary

NPTEL MOOCs Programming in C++ Partha Pratim Das 2

We have talked about the basic requirement of inheritance and we have touched upon; we have discussed the very core two concepts of the affects of inheritance between two classes, that is how does the data members are inherited and object layout done and how does the member functions can be inherited with overriding and over loading playing a role. In the current module, we will try to talk about the protected access specifiers, the semantics of the access specifier and what happens to the construction destruction process and relate that to the issue of object lifetime in the context of specialization scenarios.

(Refer Slide Time: 01:24)

The slide title is "Module Outline". On the left, there is a sidebar with a logo, the text "Module 23", "Partha Pratim Das", "Objectives & Outline", "Inheritance in C++", "protected Access", "Constructors & Destructors", "Object Lifetime", and "Summary". Below the sidebar is a circular portrait of Partha Pratim Das. The main content area is titled "Module Outline" and contains the following bullet points:

- ISA Relationship
- Inheritance in C++
 - Semantics
 - Data Members and Object Layout
 - Member Functions
 - Overriding
 - Overloading
 - **protected Access**
 - Constructor & Destructor
 - Object Lifetime
- Example – Phone Hierarchy
- Inheritance in C++ (**private**)
 - Implemented-As Semantics

At the bottom of the slide, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das". There is also a small number "3" in the bottom right corner.

The outline as I have already explained is for the whole of this inheritance and the blue part relates to what we will be discussing in this current module.

(Refer Slide Time: 01:41)

The slide title is "Inheritance in C++: Semantics". On the left, there is a sidebar with a logo, the text "Module 23", "Partha Pratim Das", "Objectives & Outline", "Inheritance in C++", "protected Access", "Constructors & Destructors", "Object Lifetime", and "Summary". Below the sidebar is a circular portrait of Partha Pratim Das. The main content area is titled "Inheritance in C++: Semantics" and contains the following bullet points:

- Derived ISA Base
- Data Members
 - Derived class **inherits** all data members of **Base** class
 - Derived class may **add** data members of its own
- Member Functions
 - Derived class **inherits** all member functions of **Base** class
 - Derived class may **override** a member function of **Base** class by **redefining** it with the **same signature**
 - Derived class may **overload** a member function of **Base** class by **redefining** it with the **same name**, but **different signature**
- Access Specification
 - Derived class **cannot access private** members of **Base** class
 - Derived class **can access protected** members of **Base** class
- Construction-Destruction
 - A **constructor** of the **Derived** class **must first** call a **constructor** of the **Base** class to construct the **Base** class instance of the **Derived** class
 - The **destructor** of the **Derived** class **must** call the **destructor** of the **Base** class to destruct the **Base** class instance of the **Derived** class

At the bottom of the slide, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das". There is also a small number "4" in the bottom right corner.

This is for your quick reference that, these are the broad semantic requirements and these two we have already done and this is the one that we will be covering in now.

(Refer Slide Time: 02:01)

Inheritance in C++:
Access Members of Base: protected Access

- Derived **ISA** Base
- Access Specification
 - Derived class *cannot access* private members of Base class
 - Derived class *can access* public members of Base class
- **protected** Access Specification
 - A new **protected** access specification is introduced for Base class
 - Derived class *can access protected* members of Base class
 - No other class or global function *can access* protected members of Base class
 - A **protected** member in Base class is like **public** in Derived class
 - A **protected** member in Base class is like **private** in other classes or global functions

NPTEL MOOCs Programming in C++ Partha Pratim Das S

Now, so what is the access specification issue? The access specification issue is as we know there are two kinds of access specifiers; the private and the public. So, for any class we have private members data as well as functions; member functions and we have public. So, this means the private members are available only to the class and the public members are available to all other classes, global functions and so on. So, with that if we have done a (Refer Time: 02:47) specialization, then the derived class, certainly cannot access the private members of the base class because the derived class is, by definition in another class. So, it is outside of the base class, it cannot access the private members. So, it can access only the public members.

Now, as you will find that the whole advantage that we want to take by modeling inheritance, by modeling the generalization specialization hierarchy, by introducing features of overriding member functions and so on will be lost, if the derived class cannot access any part of the base class, which the base class protects from everyone else. So, that has given rise to defining a new kind of access specifier known as protected.

So, the protected has a special semantics that a derived class can access the protected members of the base class. But, no other global function, no other class or global

function can access the protected members. So, what it means that, if I look at a protected member of a base class, then for the derived class it is like a public access and for other classes and global functions it is like a private access. So, we are kind of trying to; the attempt has been to create a third kind of visibility or access specification beyond private and public, so that private is totally inside the class, public is totally for everybody and protected is something that is for the class, but for some of the classes with home, this class has a very close relationship of specialization generalization or parent child kind of.

(Refer Slide Time: 05:17)

Inheritance in C++: protected Access

private Access

```
class B {
private: // Inaccessible to child
    // Inaccessible to others
    int data_;
public:
    // ...
    void Print() { cout << "B Object: ";
        cout << data_ << endl;
    }
};

class D: public B { int info_; 
public:
    // ...
    void Print() { cout << "D Object: ";
        cout << data_ << ", ";
        cout << info_ << endl;
    }
};

B b();
D d(1, 2);

b.data_ = 8; // Inaccessible to all
b.Print();
d.Print();

* D::Print() cannot access B::data_, as it is
private
```

protected Access

```
class B {
protected: // Accessible to child
    // Inaccessible to others
    int data_;
public:
    // ...
    void Print() { cout << "B Object: ";
        cout << data_ << endl;
    }
};

class D: public B { int info_; 
public:
    // ...
    void Print() { cout << "D Object: ";
        cout << data_ << ", ";
        cout << info_ << endl;
    }
};

B b();
D d(1, 2);

b.data_ = 8; // Inaccessible to others
b.Print();
d.Print();

* D::Print() can access B::data_, as it is
protected
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 6

So, let us look into example. Again, please look at the left side, do not look at the right. So, we have a class B, which has a private data member and a public method. The public method simply prints the data member. We have a derived Class D of B, which again has its own private data which is the additional data and it has a public method print. So, having understood the inheritance of member functions, we will realize that this is a case of overriding member function that is occurring here because both these print functions have the same signature.

So, D as it is inherits this, but then it overwrites and declares it again and possibly gives it a different semantics, whatever is the body here and the body here is different and this

is what we have. And then what we want to do is to instantiate these object and invoke print from both these objects that is the basic target. Now, let us look into the access issues. This is; that the data member of Class B is private. Therefore it is inaccessible to all including the child class and including the derived class, it is not accessible. So, if I want to write a print function in Class D which prints the whole of Class D, how will the class D look like? Class D will look like a layout of this is.

We have already discussed this, there is a base part which has this `data_`, there is derived part which is `info_` and that is the class. So, if I want to print this class, then I actually have an `int B :: data_`, member in this class which is this member. So, to be able to print this, I must print this as well as the info which is the member of this class. The question is that how do I access this data? This data is a private member of this class B. So, if I make it explicit when I talk about this data, I am actually, if I fully qualify applying the namespace rules, then I am actually talking about this element `B :: data_` which has been inherited by the inheritance rule of specialization.

So, accessing this data is basically accessing this data which by rule is not accessible to anyone else. Therefore, this print function cannot compile, it cannot work. Certainly, if outside if I just write `b.data_`, it will not compile, it is not accessible for the outsiders. So, this is the core problem that the protected access specifier is trying to address. Now, if D cannot access this, then there is no fun of having this data member inherited in this part of this class. So, that will mean that every time I may inherit, the base class will have an instance as a part of this object but, that cannot be read or written by the derived class.

So, it is kind a dead part in the object layout. Now for that, if we make this public, if you move this here, then naturally this problem will get solved. Print will be able to compile this but then it breaks the encapsulation because anyone else will also be able to come and read and change the data. So, we need to provide a specification which is in between these two extremes and, that is where we now introduce a protected specifier. Earlier this was private, now this is protected, so, `int data` is protected. So, nothing else changes in this code, when now in the print member function of class D, we try to refer to data which again is `B :: data_`, is this member.

This access is allowed because being protected is accessible to the child. But just look at this, this has not violated. the restriction on the other use this is written some were else in some other global function or in some other class, but that will not that access will not work because wherever I written this is not a part of a derived class of B. So, had we made this public then this would have work, but this would also have become accessible, but making this protected we had made sure that this will work, but this will continue to be a compilation error this access will not be allowable for here. So, that is the basic meaning of the protected access specifier.

Here, I am showing it in terms of data members, exactly the same semantic will work what members function as well. So, any derived class would be able to therefore, access both the public as well as protected members of the base class. So, now, with this we will be able to implement compile and also invoke the both print functions, one for this, this is simply for the B class members and this as we have over written and this is for the D class where the base part will first be printed and then the derive class part will also be printed. So, that is the basic notion of private access specifier and we will often find a good use of this happening.

(Refer Slide Time: 12:52)

```

Module 23
Partha Pratim Das
Objectives &
Outline
Inheritance in C++
protected Access
Constructors &
Destructors
Object Lifetime
Summary


## Inheritance in C++: Streaming



Streaming in B



```

class B {
protected: int data_n;
public:
 friend ostream <<(ostream& os,
 const B& b) {
 os << b.data_n << endl;
 return os;
 }
};

class D: public B { int info_n;
public:
 //friend ostream <<(ostream& os,
 // const D& d) {
 // os << d.data_n << endl;
 // os << d.info_n << endl;
 // return os;
 //}
};

B b(0);
D d(1, 2);

cout << b; cout << d;

```



B Object: 0  
B Object: 1



• d printed as a B object: info_n missing



Streaming in B & D



```

class B {
protected: int data_n;
public:
 friend ostream <<(ostream& os,
 const B& b) {
 os << b.data_n << endl;
 return os;
 }
};

class D: public B { int info_n;
public:
 friend ostream <<(ostream& os,
 const D& d) {
 os << d.data_n << endl;
 os << d.info_n << endl;
 return os;
 }
};

B b(0);
D d(1, 2);

cout << b; cout << d;

```



B Object: 0  
B Object: 1 2



• d printed as a D object as expected


```

In this connection, let me show some more examples. So, here I am just showing the use

of; further use of the protected members, here is the protected members. So, if you look into this class then you will find that; let me put it clearly. If you look into this class it has protected data member which is shielded from outside, but certainly can be used by the derive class and this class has some friend function which is trying to print that class.

Now, interestingly this being a friend function is as it is not available to class D because this function as we know actually is a global function it is just a friend. So, that it can have the visibility, because this a friend function it has the access to private data members of the class B, but it will also that access would also be extended to the protected data members of class B because being a friend this overloaded operator function has complete access within the inside of the class B. Now, if you and assuming that you do not have you are not written say this operator function in class D and you have instantiated the two objects here, naturally B needs one data for this data part, D objects needs two, this will become the `data_`; this is a base class one and this will become the `info_`.

Now, if you try to do this, what will be your expectation? Certainly your expectation would be that if I do `cout << b` this is the invocation of this function. That is the `cout << b` is basically operator output string, first member `cout` second member `b`. So, this should invoke that operator and print the B object which does connecting, but the interesting fact is that what happens, if you also write this. What is this? This is operator `d`, this `d` is of type class D and not of this type. So, you would expect your first reaction would be there should be a compilation here, but interestingly it is not. It is not a compilation here this will also go through, what it will do is and that is very, very risky and that is the reason I brought up this an example is the moment you start using protected and start using friend function you will have this.

What it will do, it will invoke this function itself, but it needs `b`, but it has `d`. How will it work, recall the layout, the layout is `d` object has two parts; one is the member it has defined which is `info_`, other is an instance of the base class object. So, what is happening here, you require a base class object at this point and you actually have a derive class object in hand, but a derive class object also has the base class part. So, what the system does, system simply ignores the additional part. It simply takes that as if this

is the base class object which you wanted to pass and lets this function be called. That is a reason just look at the result very carefully this gets printed b object value 0 gets printed from this cout, which is correct because I created object b with the value of data which is 0; clear enough.

This also prints; that is the b object and a value is 1 which is basically, this value 1 is basically the value of the B :: data_ inherited member that we are setting at this point and the two which actually got set to info will not certainly not get printed because this function does not even know of the existence of this particular data member. This whole phenomenon happens because of the something known as implicit casting of which we will discuss about much later, but I just wanted to give you this glimpse of this example to sensitize you that in terms of the; in terms of actually overloading friend functions and so on. Since, friend functions take the object as a parameter and not invoke on the object like other member functions do, you do run the risk of implicit type casting and you have to be careful to protect against that.

Naturally, the solution, the problem will become alright if you just have the overloaded operator function here as well as another overloaded operator function here where you can certainly use data because the data is protected member in b. So, d can inherit as well as access this data and in that context for this object if you try to do cout this will go here, this will go here and now this is happening not by overriding this is happening because of the some to overloading because friend functions are global functions, they are not member functions there is not context of over writing here is this two function define in two different scopes are being resolved by the difference in their signature and both of them will correctly now you will have a d object which prints 1 and 2.

So, that is the basic kind of behavior that you will expect when you start exposing the inner of the object in terms of the protected data members or protected member functions.

(Refer Slide Time: 20:38)

Inheritance in C++:
Constructor & Destructor

- Derived **ISA** Base
- Constructor-Destructor
 - Derived class *inherits* the Constructors and Destructor of Base class (but in a different semantics)
 - Derived class *cannot override or overload* a Constructor or the Destructor of Base class
- Construction-Destruction
 - A *constructor* of the Derived class *must first* call a *constructor* of the Base class to construct the Base class instance of the Derived class
 - The *destructor* of the Derived class *must* call the *destructor* of the Base class to destruct the Base class instance of the Derived class

NPTEL MOOCs Programming in C++ Partha Pratim Das

Let us move on and quickly talk on the constructor/destructor. Now, when the derived class inherits the constructor and destructor of base class, but you have to be very cautious it is with the different semantics. See, if I say, it inherits then the problem is a constructor is a fixed name function; a constructor of a class has a same name as a class. So, if I inherit, if the base class constructor into the derived class then certainly it cannot behave as a constructor. So, the base constructor becomes kind of a member function in the derived class constructor, in the derived class.

Similarly, base class destructor becomes kind of a member function in the derived class and they are, but it is very important that this inheritance of the constructor are possible because again recall that any derived class object has a base class instance. This is an instance of the base. So, if I have to construct the derived class object as a whole certainly I will have to construct the base class instance part of it and that needs the base class constructor to be available. Similarly, if I will construct then I need to destruct. So, by symmetry I will require the other.

(Refer Slide Time: 22:12)

The slide title is "Inheritance in C++: Constructor & Destructor". It features a logo of a person in a graduation cap and gown. The left sidebar contains navigation links: Module 23, Partha Pratim Das, Objectives & Outcome, Inheritance in C++, Inherited Access, Constructor & Destructor, Object Lifetime, and Summary. A small video thumbnail of the speaker is on the left.

```
class B { protected: int data_a;
public:
    B(int d = 0) : data_a(d) { cout << "B::(int): " << data_a << endl; }
    ~B() { cout << "B::~B(): " << data_a << endl; }
    // ...
};

class D : public B { int info_a;
public:
    D(int d, int i) : B(d), info_a(i) // ctor-1: Explicit construction of Base
    { cout << "D::(int, int): " << data_a << ", " << info_a << endl; }

    D(int i) : info_a(i) // ctor-2: Default construction of Base
    { cout << "D::(int): " << data_a << ", " << info_a << endl; }

    ~D() { cout << "D::~D(): " << data_a << ", " << info_a << endl; }
    // ...
};

D b(B);
D d1(1, 2); // ctor-1: Explicit construction of Base
D d2(3); // ctor-2: Default construction of Base
```

Object Layout:

Object b	Object d1	Object d2
1	1 2	0 3

NPTEL MOOCs Programming in C++

So, this quickly shows what is possible, we have a base class constructor here, which the data pattern have not changed, the base class constructor there is a base class destructor, there is a base. This destructor does not do anything, simply prints a message and base class constructor is overloaded. It is a parameterized constructor and also a default constructor. So, basically there are two constructors here. For the derived class, we have added one data member. So, there are two values to set. So, what we do, we actually have provided two different constructors, one that takes two parameters and the other that takes one parameter.

So, what is interesting and new in this whole scenario is this piece of the code in the initialization list. The derive class object that needs to get constructed will need to have a base part instance of the base object. So, this needs to get constructed. Now, the basic principle is that you cannot proceed with the construction of the derive class components. There could be several components here, until you have finished with the construction of the base instance because base instance in a derive class object is unique. It may have 0 or 1 or any number of data members, but if in the under inheritance you can have only one base class instance in the object. So, that must get constructed.

Here, what the base class construct, sorry, the derive class constructor is doing? The

derive class constructor is actually calling the base class constructor, this is the name. So, this function is inherited here and the derive class constructor is actually calling this function by name passing the parameter that its needs which it once set in the data. So, the first this will happen this b data will happen and this will get set and then other initializations of its own members will happens.

What happens in this case? In this case there is no base class constructor being called that does not mean that the base class constructor will not need to be called. I have not put it explicitly here, but still at this point the base class constructor will get called because unless the base class part of the object has been constructed, the derive class object cannot get constructed. So, this will; like the compiler provides the free functions constructor destructor all of that. Similarly, this is where the compiler provides a free invocation of the base class a constructor.

Now, naturally if the compiler provides a free invocation then a compiler cannot set a parameter to the base class constructor. So, this is possible that is this kind of a constructor for derived class can be written provided the base class has a default constructor, which does not need a parameter. So, very simply if I just erase this default parameter in the base class constructor then this will become compilation error because it will say for this overloaded constructor of d, there is no default base class constructor, there is no base class constructor called. So, with this we can construct the objects, for example, if we construct is the d1 object with 1, 2 then certainly this construct is getting used 1 get sets to d, then a base class construction happens that get set to here. So, if we do this then the eventual result will be that will 1 here and 2 here for the object d1.

In case of object d2 we are just passing one parameter. So, by overloading this means that this constructor will get invoke, which means that the base class constructor will get invoke with the default as a default constructor with the default value. So, this will become 0. So, in this case this part will become 0 and 3 will go to i which will come here which will go set here. So, that is all, the object will get constructed. So, here I have shown this layout. So, you could a take different examples and try your layout and get convinced that how this constructors work between themselves and similarly that base class will need to have destructor, the derive class will need to have destructor.

(Refer Slide Time: 27:44)

The slide title is 'Inheritance in C++: Object Lifetime'. The left sidebar contains navigation links: Module 23, Partha Pratim Das, Objectives & Outcome, Inheritance in C++, Inherited Access, Constructor & Destructor, Object Lifetime, Summary. The main content area shows C++ code for classes B and D, their constructors, and an object creation sequence. Below the code is a table comparing construction and destruction output. A note at the bottom states: 'First construct base class object, then derived class object' and 'First destruct derived class object, then base class object'.

```
class B {protected: int data_;}
public:
    B(int d = 0) : data_(d) { cout << "B::B(int): " << data_ << endl; }
    ~B() { cout << "B::~B(): " << data_ << endl; }
// ...
};

class D: public B { int info_; 
public:
    D(int d, int i) : B(d), info_(i) // Explicit construction of Base
    { cout << "D::D(int, int): " << data_ << ", " << info_ << endl; }
    D(int i) : info_(i) // Default construction of Base
    { cout << "D::D(int): " << data_ << ", " << info_ << endl; }
    ~D() { cout << "D::~D(): " << data_ << ", " << info_ << endl; }
// ...
};

B b(0);
B d1(1, 2);
D d2(0);

Construction O/P
B::B(int): 0          // Obj. b
B::B(int): 1          // Obj. d1
D::D(int, int): 1, 2 // Obj. d1
B::B(int): 0          // Obj. d2
D::D(int): 0, 2       // Obj. d2

Destruction O/P
D::~D(): 0            // Obj. d2
B::~B(): 0            // Obj. d1
D::~D(): 1, 2         // Obj. d1
B::~B(): 1            // Obj. d1
D::~D(): 0            // Obj. b
```

• First construct base class object, then derived class object
• First destruct derived class object, then base class object

NPTEL MOOCs Programming in C++ Partha Pratim Das 12

Now, with that if we just try to trace this messages that we have put and in the next slide we will try to do that. Let us say, if you just trace these messages and see then will be able to find out what is lifetime of the objects now. So, let us just try to do this, this is the same construction. So, naturally when this is encountered what has to happen? The base class constructor will have to get called for object being, that is the first thing that happens this message gets printed. Then this construction has to happen, the object d 1 of class D has to; if this has to happen then this constructor gets called, which in turn means that this constructor gets called. So, the next one you see is the construction of b object with the value 1 this is what here for the d1 object that I have constructed.

Once that has been done then the construction of the d1 object by the constructor of class D will get completed. So, I will get the next message, these two is the basically the construction of d1. Next is; so, we have seen up to this point, next is the construction of object d2 which invokes this constructor, which invokes the default constructor. So, I will again have constructor b called for object d2, but the parameter value is 0. Once that has been done then the rest of the construction of this will continue and that will print the next message. You can see that the lifetime of an object continues to be same only with the exception that lifetime of the base object will start before the lifetime of the derived object can actually start. So, that will have to get constructed totally in full.

So, which means by symmetry of object lifetime that you have already seen, certainly if we look in to the remaining part of the messages as to what happens at the time of destruction that is when this object go out of scope at this point, certainly this was last object constructed. So, that with the first object you get destructed and what will happen this was last construction message. So, if you now look at the first destruction message is for the object d2, that is this message. So, this destruction message is basically this one and only when the control reaches at this point, you first clean up whatever additional members that derive class object has and only when you have reached the end having done this, then this destructor of the base class gets called, when you clean up the base class.

So, you are first constructing the base class instance then constructing the remaining data members of the derive class. On the wrap-up time first clean up the non base data members of the derive class object and then clean up the base class part. So, if you look into try to trace for d1, will see the same symmetric behavior and I would leave it to you to complete. So, this gives you the clear picture in terms of how the layout of the objects will be managed.

(Refer Slide Time: 31:25)

Module Summary

- Understood the need and use of protected Access specifier
- Discussed the Construction and Destruction process of class hierarchy and related Object Lifetime

Module 23
Partha Pratim Das
Objectives & Outline
Inheritance in C++
protected Access
Construction & Destructor
Object Lifetime
Summary

NPTEL MOOCs Programming in C++ Partha Pratim Das 14

So, to summarize on this module we have understood and really analyzed the need and

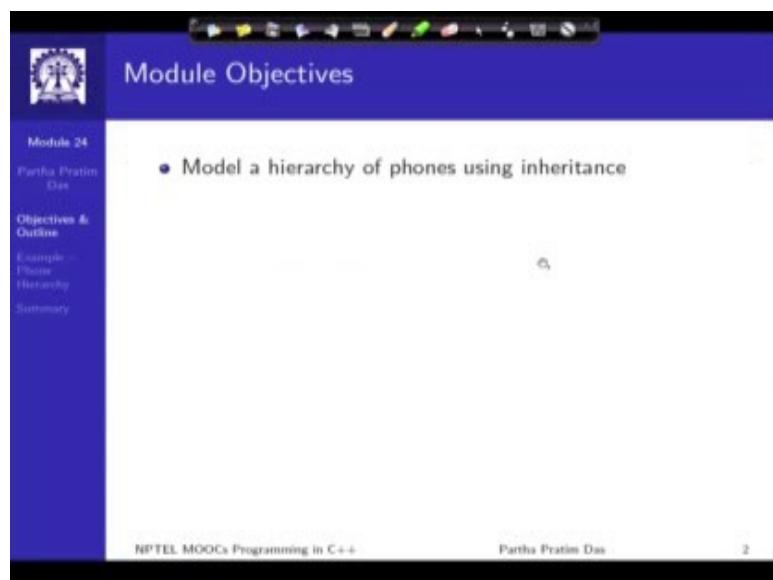
use of protected access specifiers for specifying access to different base class members, and we have seen how the construction and destruction process happens between a derived class object and a base class object and really under try to follow trace the lifetime of the derive class object with respect to the base class object.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 39
Inheritance: Part IV

Welcome to Module 24 of Programming in C++. From the last couple of modules we have been discussing about inheritance in C++.

(Refer Slide Time: 00:38)



We have understood the basic definition and structure of inheritance and in that connection in this module we would like to try to work out an example of hierarchy of phones that we use today. We would like to show during this module that, how we can actually take the abstraction of different concepts, try to create the class modules for those in C++, organize them the resulted ISA hierarchy in terms of C++ code and create possibilities for extension.

(Refer Slide Time: 01:30)

Module Outline

Module 24
Partha Pratim Das
Objectives & Outline
Example – Phone Hierarchy Summary

- ISA Relationship
- Inheritance in C++
 - Semantics
 - Data Members and Object Layout
 - Member Functions
 - Overriding
 - Overloading
 - `protected` Access
 - Constructor & Destructor
 - Object Lifetime
- Example – Phone Hierarchy
- Inheritance in C++ (`private`)
 - Implemented-As Semantics

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, in terms of the outline as I had mentioned this is the complete outline of what we are discussing in the basic level of inheritance, we have already talked about ISA relationship in modeling that OOAD frequently used and using that we have shown how to express ISA relationship in terms of two or more C++ classes, based on that we have defined detailed semantics for the inheritance of data members, the inheritance of member functions.

We have seen that when a class is derived from another base class then it inherits all the data members and member functions. And the member functions can be redefined with the same signature to overwrite them or we can introduce definitions of member functions by the existing name or inherited name with the different signature to overload that member function as we could do earlier. Further, we have seen how to add new data members, we have also taken a look into the access specification of the data and member and functions of a derived class as derived from the base class, we have introduced a new access specifier called `protected`, which has special semantics for derived classes. For derived classes all `protected` data members of the base are accessible, whereas these `protected` data members are not accessible for external functions and other classes.

(Refer Slide Time: 03:34)

The screenshot shows a presentation slide titled "Phone Hierarchy". The slide has a blue header bar with the title. On the left, there is a vertical sidebar with a logo at the top, followed by navigation links: "Module 24", "Partha Pratim Das", "Objectives & Outline", "Example - Phone Hierarchy", and "Summary". The main content area contains a bulleted list of points:

- Let us model a hierarchy of phones comprising:
 - Land line Phone
 - Mobile Phone
 - Smart Phone
- We model Helper classes
- We model each phone separately
- We model the phone hierarchy

At the bottom of the slide, there is a circular portrait of a man, identified as "Partha Pratim Das". The footer of the slide includes the text "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

We have also seen the construction and destruction process and the object life time. Using all those notions we would now move into discussing a creation of a basic module structure for a set of phones. So, we start with a model hierarchy of phones and we just start simple assuming that there are three kinds of phones that make our world. The landline phones, the typical ones that we had been using maybe 20 years back, 30 years back. Then the mobile phones, the so called nowadays what is more frequently called as a dumb mobile phone, which can do only restricted functions and the all pervading variety of smartphones that has invaded our life.

So, in the process of doing that, we will first model a set of helper classes, these helper classes will allow us to have different types for the attributes that we will need to handle then we will model each phone separately. For each of these, we will try to write down an outline of a class description and then we will see that given that, these phones have certain inherent hierarchy of specialization, how do we combine this code into a C++ inheritance hierarchy and how does that simplify the total model.

(Refer Slide Time: 05:06)

The screenshot shows a presentation slide titled "Helper Classes". On the left, there is a sidebar with a logo, the title "Module 24", and navigation links: "Partha Pratim Das", "Objectives & Outline", "Example - Phone Hierarchy", and "Summary". The main content area has a table with two columns: "Class" and "Description".

Class	Description
<code>class PhoneNumber</code>	12-digit phone number
<code>class Name</code>	Subscriber Name (as string)
<code>class Photo</code>	Image & Subscriber Name as alt text
<code>class RingTone</code>	Audio & ring tone name
<code>class Contact</code>	<code>PhoneNumber</code> , <code>Name</code> , and <code>Photo</code> (optional) of a contact
<code>class AddressBook</code>	List of contacts

In the bottom right corner of the slide, there is a circular video player showing a person speaking.

So, to start with we first have a set of helper classes. So, I have not included the details of these classes, you can work those out. One class naturally that we need is to represent the phone number certainly and phone numbers as we know that in India, the phone numbers are 12 digit including the country code. So, it is sum type, which presents 12 digit numbers. Then certainly we need the name of the subscriber, who is actually subscribing to the phone, we need the name also for our contacts, the people whom we want to call.

We have provision for having photos of our contacts, so will have some helper class as photo which will have image and the alternate text for the image and so on. We will have a class defining ringtones, the audio file and the name of that ringtone, etcetera. We will have helper class for contact, the minimum information of a contact is a phone number and name and optionally we could have a photo of that contact also. Of course, there are several other that are possible, like the designation, like company and so on. We are just keeping it simple right now. Finally, we will have a helper class address book which is a collection of maybe set or list of contacts, which we make and maintain in our phone. So, these were the simple set of basic helper classes let us go ahead with the design.

(Refer Slide Time: 06:52)

Land line Phone Model

- Landline Phone
 - Call: By dial / keyboard
 - Answer

```

class LandlinePhone {
    PhoneNumber number_;
    Name subscriber_;
    RingTone rTone,i;
public:
    LandlinePhone(const char *num,
                  const char *sub);
    void Call(const PhoneNumber *p);
    void Answer();
    friend ostream << (ostream& os,
                        const LandlinePhone& p);
};

```

NPTEL MOOCs Programming in C++
Partha Pratim Das

So, first we want to model the landline phone. When we want to model the landline phone, we identify that if I have a landline phone; however, primitive you can think about those black heavy sets with rotating dial that, we used to use several years ago. The basic functionality that a landline phone must support is an ability to call and an ability to be called back. So, we have call functionality and we have an answering functionality, without that phone is not defined. So, if I have that, then as we go on to designing the class for this phone we are trying to create this model. Suppose, I have given it a name land line phone and for the call functionality, I have introduced a call method, call member function in the class and to be able to call we need the number that we have to call. So, the phone number will be a parameter to this call member function.

Similarly, if a call arises then we have to answer that call so, there is a member function answer introduced which will answer the phone call. In terms of the data members, what do we need certainly every landline phone has a number of its own, the number at which it can be called. So, we will have a number member the phone must have been subscribed by someone so, that is a subscriber's name and we may have a ringtone, which for very old phones are fixed ringtones of what we typically would say like crink crink sound of the phone, buzzing sound of the phone.

Based on this naturally, we need the constructor to be defined, which will at least take the name and the number and the subscriber. These two members have to be initialized

for any meaningful construction of the phone, the initialization of this ringtone we could keep optional, which say that by default it has some specific ringtone. Now, to end we have also would introduce an overloading in terms of the friend operator function for outputting the information about this particular landline phone objects. This is required more for debugging and programming trace purposes then actually the phone functionality. So, it is an auxiliary functionality, which we add in terms of a friend function to be able to work with this particular class.

(Refer Slide Time: 09:59)

Mobile Phone Model

- Mobile Phone
 - Call: By keyboard – shows number
 - By Number
 - By Name
 - Answer
 - Redial
 - Set Ring Tone
 - Add Contact
 - Number
 - Name

```

class MobilePhone {
    PhoneNumber number_;
    Name subscriber_;
    RingTone rTone_;
    AddressBook abook_;
    PhoneNumber *lastDialled;
    void SetLastDialled(const PhoneNumber& p);
    void ShowNumber();
public:
    MobilePhone(const char *num,
                const char *sub);
    void Call(PhoneNumber *p);
    void Call(const Name& n);
    void Answer();
    void Redial();
    void SetRingTone(RingTone::RINGTONES r);
    void AddContact(const char *num = 0,
                    const char *sub = 0);
};

friend ostream &operator<<(ostream& os, const MobilePhone& p);
}

```

NPTEL MOOCs Programming in C++

Partha Pratim Das

Let us move on, let us now talk about the mobile phone. Certainly, the mobile phone will have a call functionality, will have an answer functionality, In terms of the call functionality, usually in the mobile phone, we would either be able call the number as we could do in a landline phone, but it is also possible that the mobile phone will have some kind of a address book mechanism, so that I can keep a set of contacts and I can pick up some contact by the name and I can call that contact by that name. So, in terms of call we do see, in terms of mobile phone two options, which one of which that is calling by name was not available in the landline phone.

In addition to that we have usually we have an option to redial, typically to redial the last number that was called and in a majority of mobile phones we also have the option to set

a ringtone. So, this of course there are several other that we can think off, but certainly by mobile phone as I mentioned I am talking about the cellular phones of kind of the early generations like, typically many of you may have seen and used Nokia 1100 series kind of phones where you just have ability to be mobile, but you do not have all those different fancy functionalities that we see today.

Let us see for this simple mobile phone class how do we make the definition, so we have created the class with the name mobile phone for each of the functionality that, we see they will have to be some member functions associated. So, for call we will have a member function, this looks pretty much like the call member functions we are done for the landline phone class, so which takes a phone number and makes the call, but now we see that there is additional functionality that, I can call someone by name. So, we will keep another call member function which takes the name of a person and makes the call.

So, we can see that here because of the duality of the call function, we are having to introduce overloaded member functions. We will have an answer, which is the answer member function. We introduce a redial member function to be able to redial a member function. To set the ringtone which certainly will have to take the particular ringtone that I want to set and for adding new acquaintances to my address book, I need a add contact where, I need to specify the number and I need to specify the name of the person. So, with this we have so, as we start after deciding the name of the class, these are first things that we complete which will give us the interface, as we say of what this class should be doing that is a basic functionality of the class, the set of operations for the class.

Having specified that, now we look into the internals of the classes, if I have to support these member functions, if I have to support the functionality that I want, what are the data members that I will need? Certainly so, looking into that certainly I need the number of the phone which is as before, I need a subscriber name to reach this particular mobile phone is been subscribed and being mobile phone in all likelihood, it will have the possibility of setting different ringtones so I have a ringtone members. So, these are pretty much like what we had seen before, but now we have assumed that it is possible that we keep the context in our phone. So, I need an address book. So, a book is a

member which will keep, which is an address book that is it will keep a list or set of contacts that I would like to refer to at often times.

So, if I add do an add contact that basically will add the contact here and I would also need to need a feature, I have a provided an interface that, I would like to redial, just simply redial the number that I had dialed last. So, I need a member to remember, what is the last number that I had dialed? So, that brings in this data member, besides that I may need some of the; so these are the basic data members that I will require to support this functionality and in addition I will need some more member functions, for example, if I think about let us say redial.

(Refer Slide Time: 15:41)

Mobile Phone Model

Module 24
Partha Pratim Das
Objectives & Outline
Example - Phone Hierarchy
Summary

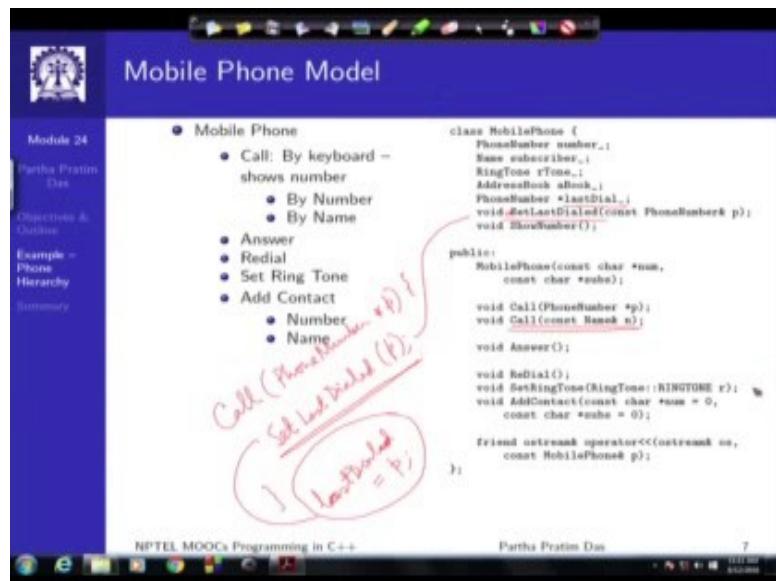
- Mobile Phone
 - Call: By keyboard – shows number
 - By Number
 - By Name
 - Answer
 - Redial
 - Set Ring Tone
 - Add Contact
 - Number
 - Name

```
class MobilePhone {  
    PhoneNumber number_;  
    Name subscriber_;  
    RingTone rTone_;  
    AddressBook abook_;  
    PhoneNumber *lastDial_;  
public:  
    void SetLastDial(const PhoneNumber& p);  
    void ShowNumber();  
  
    MobilePhone(const char *nm,  
               const char *sub);  
  
    void Call(PhoneNumber *p);  
    void Call(const Name& n);  
  
    void Answer();  
  
    void Redial();  
    void SetRingTone(RingTone::RINGTONES r);  
    void AddContact(const char *nm = "",  
                  const char *sub = "");  
  
    friend ostream << (ostream& os,  
                      const MobilePhone& p);  
};
```

Redial(?)
Call(last Dial)

So, if I think about redial, if I want to redial then I will need to give a call to; this is the last dial. So, I will need to give a call to this member function. So, I will have to do call, redial that will be last dial and that will be my basic redial functionality. So, I need a way to set this data member, I need a way to remember this data member. So, what does it mean? It means that when I am dialing some number, I am calling some number I must remember that.

(Refer Slide Time: 16:25)



So, if I look into the call functionality of the call member function, say for phone number* type then what I will need is to set this particular last dial function; last dial number. So, here I include this member function which I will invoke from my call method. Similarly, when I implement the other call, overloaded call method also I will have invoked this, which will set the last dialed number. Now, you may wonder as to, am I setting up separate member function to do that? The other option could have been instead of doing this I could have simply done last dialed or last dial is assigned. I could have simply set this to p because it is all in the domain of all in the same namespace of the mobile phone class.

The reason I perceive that I will rather have a member function to start with because it is possible that when I set this last dial number I may want to keep some more properties along with the just the last dial number, for example, I might want to also remember as to when did I dial this person the last time. I may want to also remember as to what was fate of the last call, did it go through or was it missed and if it did go through then what is a duration for which I took the call and so on. The actual functionality could be extended, extensible in several different ways. So, it may be lot more than just setting this last dial number.

So, I try to model that in terms of a member function. So, that is a typical style that we will often try to follow that, whenever we want to set some member, data member or may be at times get some data member even from within the member function of the class, we may want conceive whether we would directly set that or we will use some private member functions to set them. So that if there are additional functionalities then those functionalities can also be put through.

(Refer Slide Time: 19:21)

The slide is titled "Mobile Phone Model". On the left, there is a sidebar with navigation links: Module 24, Partha Pratim Das, Objectives & Outline, Example - Phone Hierarchy, and Summary. The main content area has two columns. The left column lists features of a mobile phone:

- Mobile Phone
 - Call: By keyboard – shows number
 - By Number
 - By Name
 - Answer
 - Redial
 - Set Ring Tone
 - Add Contact
 - Number
 - Name

The right column contains the C++ code for the `MobilePhone` class:

```

class MobilePhone {
    PhoneNumber number_1;
    Base subscriber_1;
    RingTone rTone_1;
    AddressBook abook_1;
    PhoneNumber lastDial_1;
    void SetLastDial(const PhoneNumber& p);
    void ShowNumber();
public:
    MobilePhone(const char *num,
                const char *name);
    void Call(PhoneNumber *p);
    void Call(const Name& n);
    void Answer();
    void Redial();
    void SetRingTone(RingTone::RINGTONES r);
    void AddContact(const char *num = 0,
                    const char *name = 0);
    friend ostream &operator<<(ostream& os,
                                const MobilePhone& p);
};


```

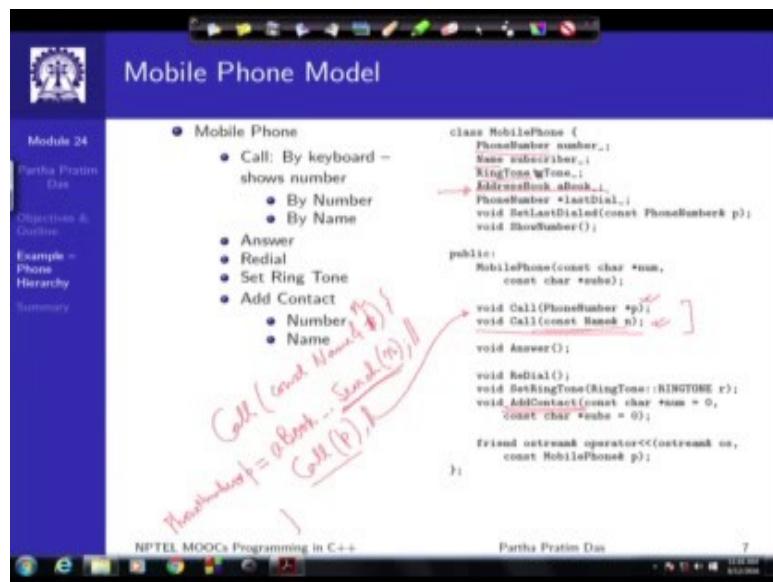
A small video window in the bottom right corner shows a person speaking. There are handwritten notes in red ink on the slide, pointing to the "SetLastDial" and "ShowNumber" methods in the code.

Now, naturally if we perceive it that way then we need this member function to be present in the mobile phone class and the question is this member function should it be in public, answer is no because we have already defined the interface. This is what we want the interface to be that is what others external classes and global function should be able to access because that what is a view of the phone that you have. So, if you think about a physical phone, you have options for doing all of these interface activities on the keyboard or the graphics display, your touch screen, the phone, but do you really see any functionality where you can set the last dial number, you do not because you are not supposed to do that.

That is internal to the phone and therefore, such member functions are private and as we say this is for the interface, this member functions are for the implementation. So, you

will in the design you will need to keep this distinction clearly in mind. Similar to that I may have another member function as show number that when the dialing has happened when you are calling someone you might want to show the number that your calling or when a call has come the phone ringing and you want to answer you would like to see the number that is calling you.

(Refer Slide Time: 20:59)



So, the ShowNumber() is a member function which is supposed to do all this behavior. There will be several such additional member functions that you would need, but I just, I am illustrating two such to explain to you that there could be several member functions in the private part of the class as well, which are basically supporting functions for realizing the interface and other supporting functions, which you do not want to make feasible in the public space. Similarly, we may talk a little bit about this call, which basically Call() a person by name.

So, if I have this then what are the functionalities that we will need? Certainly, we need to, I am sorry, this should be n. So, I need to actually know the number this person has. So, I would assume that, why do I get the number certainly the number has to come from the address book. So, I would have to assume that address book will have some kind of Search() member function such method which given a name finds out and returns me the

phone number. So, I can have the phone number return in terms of that and once that have been returned then I can use the other interface function which can make a call based on a phone number.

We can use that to realize the actual functionality of this call because if you look between these two overloaded member functions, the basic functionality is the call which is realized by the first one and the added to your functionality is search person you want to call and then make a call that is what is realize by the second function here. So, this is where you write the code to search and then you actually make the call and that is how you should go ahead with doing design and as you go ahead your finding that I said that I will not give you the details of the helper classes because their interfaces, their member functions will kind of get derived from your design because we have just seen this requirement for an address book. We have seen the requirement to adding a contact the address book and so on. So, all this will derive the different interface methods that the address book class should have. By similar reasoning you should be able to find the different interface requirements of the other helper classes as well. So, we have fairly detail description of the mobile phone class.

(Refer Slide Time: 24:05)

The slide is titled "Smart Phone Model". On the left, there's a sidebar with navigation links: Module 24, Partha Pratim Das, Objectives & Outline, Example - Phone Hierarchy, and Summary. The main content area has a blue header with the title. Below it, there's a bulleted list of features for a Smart Phone, followed by a class definition for SmartPhone.

```

class SmartPhone {
    PhoneNumber number_1;
    Name subscriber_1;
    RingTone rTone_1;
    AddressBook abook_1;
    PhoneNumber *lastDial_1;
    void SetLastDial(const PhoneNumber p);
    void ShowNumber();
    unsigned int size_1;
    void DisplayPhoto();
public:
    SmartPhone(const char *num,
               const char *sub);
    void Call(PhoneNumber *p);
    void Call(const Name n);
    void Answer();
    void Redial();
    void SetRingTone(RingTone::RINGTONES r);
    void AddContact(const char *name = 0,
                    const char *sub = 0);
    friend ostream &operator<(ostream &os,
                                const MobilePhone &p);
}
  
```

A red bracket is drawn from the word "Answer" in the list of features to the "Answer()" method in the code. A red arrow points from the "Redial" method in the list to the "Redial()" method in the code. There is also a red mark near the "Answer()" method.

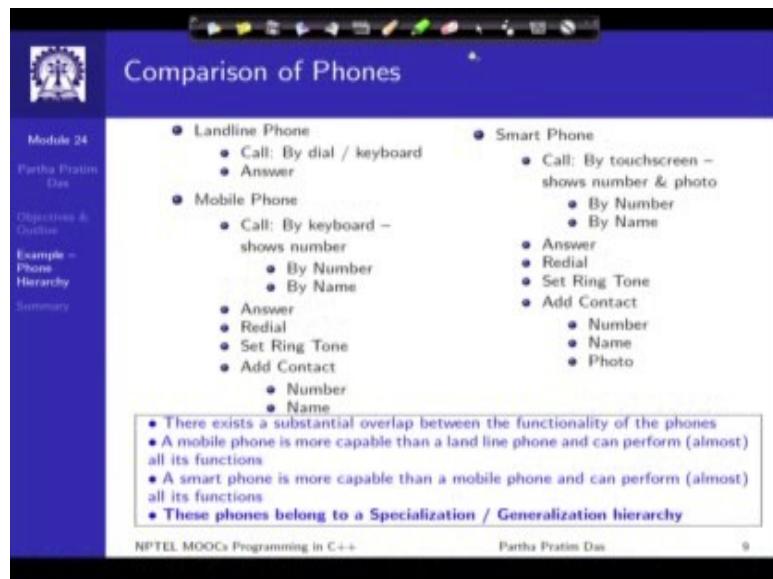
At the bottom of the slide, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

So, let us move on to the smart phone with respect to the mobile phone, I kept the smart phone relatively simple; it can call by touch screen. The smart phones typically are characterized by having a touchscreen. So, earlier I was calling from the mobile phone with a keypad. Now, I am calling with a touch screen, the basic functionality remains the same. So, I am now showing the design requirements and the design of the class together. So, the basic requirement remains the same that I call a number or I call a name, but I would again need to have both of these, these are overloaded, but what you should now start realizing that these are situations of possibilities of overriding that is arising here because I have the same functionality of been able to call a phone number, but in that functionality what is changing is the implementation of this function.

Earlier it was by keypad, now it is by the touch screen. I have answer, redial these are all like what we discuss for mobile phone. In terms of AddContact(), now it thus has a touchscreen smart display. So, I will need to probably would like to have photos of my contacts also to be added. So, the interface looks pretty much similar though it may need reimplementation, these data members are the same as what we did in mobile phone. These methods, private methods are also same, but I may have additional data members for example, if it is a smart phone then one major criterion of finding a smart phone is a size of the display, is it a 7 inch phone, it is a six and half inch phone and so on.

So, I might want to put the size, when I am making a call or I am answering a call I would like to see the face of the person whom I am calling or who is calling me provided that person is in my address book. So, possibly I will have a functionality like Display Photo() as a private member function in smart phone, there could be several others as well. So, these are basic class descriptions of the landline, mobile and smart phones.

(Refer Slide Time: 26:43)



The slide title is "Comparison of Phones". It contains two columns of bullet points comparing Landline Phone and Smart Phone, followed by a summary section.

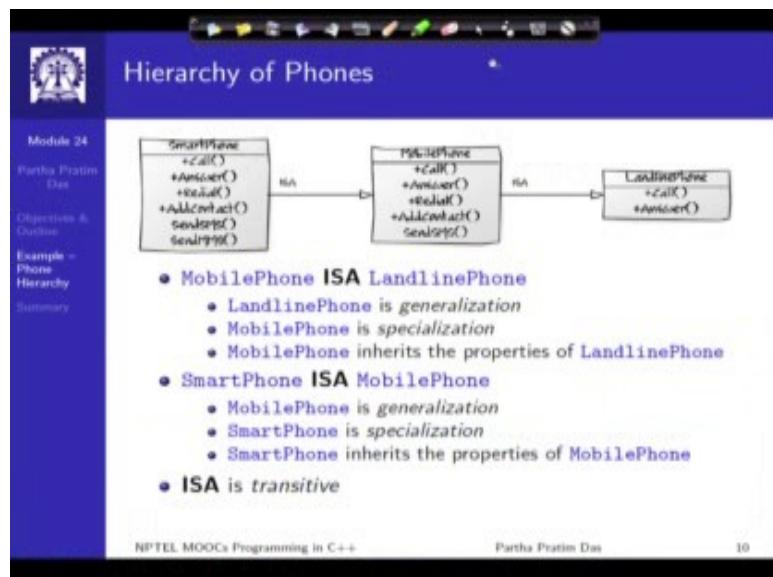
<ul style="list-style-type: none">● Landline Phone<ul style="list-style-type: none">● Call: By dial / keyboard● Answer● Mobile Phone<ul style="list-style-type: none">● Call: By keyboard – shows number<ul style="list-style-type: none">● By Number● By Name● Answer● Redial● Set Ring Tone● Add Contact<ul style="list-style-type: none">● Number● Name	<ul style="list-style-type: none">● Smart Phone<ul style="list-style-type: none">● Call: By touchscreen – shows number & photo<ul style="list-style-type: none">● By Number● By Name● Answer● Redial● Set Ring Tone● Add Contact<ul style="list-style-type: none">● Number● Name● Photo
--	--

Summary:

- There exists a substantial overlap between the functionality of the phones
- A mobile phone is more capable than a land line phone and can perform (almost) all its functions
- A smart phone is more capable than a mobile phone and can perform (almost) all its functions
- **These phones belong to a Specialization / Generalization hierarchy**

So, this is the summary of their different functionality and as we had seen before that there is a strong sense of generalization specialization that exists amongst these concepts among these classes.

(Refer Slide Time: 26:59)



The slide title is "Hierarchy of Phones". It shows a class hierarchy diagram with three classes: SmartPhone, MobilePhone, and LandlinePhone. SmartPhone is at the top, followed by MobilePhone, and then LandlinePhone at the bottom. Each class has a list of methods. Below the diagram, a bulleted list explains the relationships:

```
graph TD; SmartPhone --> MobilePhone; MobilePhone --> LandlinePhone;
```

- **MobilePhone ISA LandlinePhone**
 - LandlinePhone is *generalization*
 - MobilePhone is *specialization*
 - MobilePhone inherits the properties of LandlinePhone
- **SmartPhone ISA MobilePhone**
 - MobilePhone is *generalization*
 - SmartPhone is *specialization*
 - SmartPhone inherits the properties of MobilePhone
- **ISA is transitive**

So, we can quickly conclude that here we have mobile phone ISA landline phone and smart phone ISA mobile phone and with that now, we can look at the total functionality.

(Refer Slide Time: 27:15)

```
class LandlinePhone {
    PhoneNumber number_1;
    Name subscriber_1;
    RingTone rTone_1;

public:
    LandlinePhone(const char *num,
                  const char *subs);
    void Call(const PhoneNumber &p);

    void Answer();
};

friend ostream << ostream& os,
               const LandlinePhone& p);
}

class MobilePhone {
    PhoneNumber number_1;
    Name subscriber_1;
    RingTone rTone_1;
    AddressBook abook_1;
    PhoneNumber *lastDial_1;
    void SetLastDial(const PhoneNumber& p);
    void ShowNumber();

public:
    MobilePhone(const char *num,
                const char *subs);
    void Call(PhoneNumber &p);
    void Call(const Name n);
    void Redial();
    void Answer();
    void SettingTone(RingTone::RINGTONE r);
    void AddContact(const char *name = "",
                    const char *num = "");
};

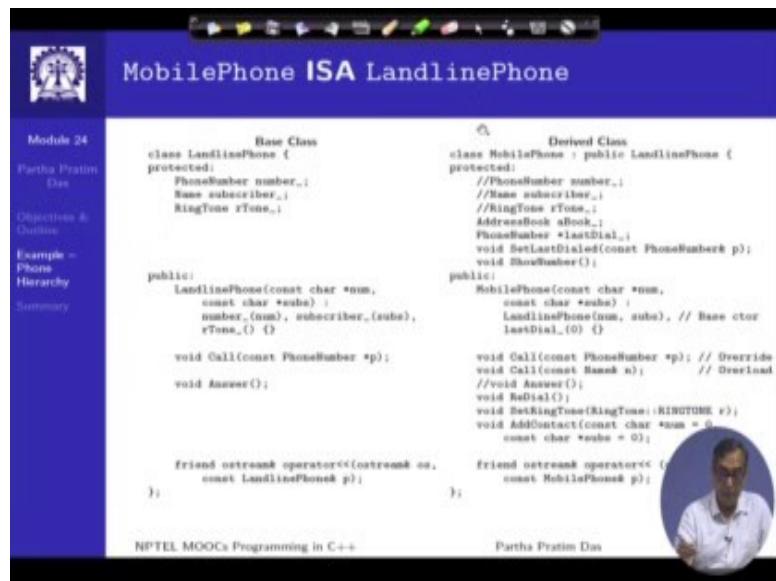
friend ostream << ostream& os,
               const MobilePhone& p);
}
```

That, if we now place that design of the landline phone class and the mobile phone class and look at them side by side then we see that there are several data member which have basically common, but there are others which are new. Similarly, there are some methods which are common, there some methods need new signature and there several other methods which are new. So, with this observation we can plan actually to combine them in terms of a hierarchy and reduce minimize the design that we have.

(Refer Slide Time: 27:59)

These are just to show you in terms of how does it look in terms of the UML model? This is for your further understanding.

(Refer Slide Time: 28:05)



The screenshot shows a slide from a presentation titled "MobilePhone ISA LandlinePhone". The slide contains C++ code illustrating inheritance. On the left, there's a sidebar with navigation links: Module 24, Partha Pratim Das, Objectives & Outcome, Example - Phone Hierarchy, and Summary. The main content area displays two classes:

```
Module 24
Partha Pratim Das
Objectives & Outcome
Example - Phone Hierarchy
Summary

MobilePhone ISA LandlinePhone

Base Class
class LandlinePhone {
protected:
    PhoneNumber number_;
    Name subscriber_;
    RingTone rTone_;
public:
    LandlinePhone(const char *num,
                  const char *sub) :
        number_(num), subscriber_(sub),
        rTone_() {}
    void Call(const PhoneNumber *p);
    void Answer();
    friend ostream operator<<(ostream& os,
                                 const LandlinePhone& p);
};

Derived Class
class MobilePhone : public LandlinePhone {
protected:
    PhoneNumber number_;
    Name subscriber_;
    RingTone rTone_;
    AddressBook aBook_;
    PhoneNumber *lastDial_;
public:
    MobilePhone(const char *num,
                const char *sub) :
        LandlinePhone(num, sub), // Base ctor
        lastDial_(0) {}
    void Call(const PhoneNumber *p); // Override
    void Call(const LandlinePhone *p); // Overload
    void Answer();
    void Redial();
    void SetRingTone(RingTone::RINGTONES r);
    void AddContact(const char *num = 0,
                    const char *sub = 0);
    friend ostream operator<<(ostream& os,
                                 const MobilePhone& p);
};

NPTEL MOOCs Programming in C++
Partha Pratim Das
```

But now, I can just use this observation to model not just the mobile phone and the landline phone separately, but I can model as mobile phone ISA landline phone. So, I introduce the inheritance here, the moment I introduce inheritance here the need for these data members do not exist because they exist in the base class and will automatically get inherited, these members certainly are added. Similarly, when I do this I if I perceive as we have discussed that we did not have any specific difference in terms of way the answering is done, we just pick up the call and start talking.

So, I might perceive that the way you answer in a landline phone and the way you answer in a mobile phone are the same. So, this commented out which means that this particular function will get inherited from the landline phone, the parent class, but in terms of the call we need to still define this signature that we need to override because the way to call would be very different, for example, a landline phone I may just be using dial in a mobile. I am using a certainly I am not using a dial I am certainly using some kind of a keypad keyboard and further I need another version of the Call() function as we I has mentioned and which will mean that I have a overload here. So, with this the design of the mobile phone class gets further simplified.

(Refer Slide Time: 29:46)

The screenshot shows a code editor window titled "SmartPhone ISA MobilePhone". The code is a class hierarchy:

```
Module 24
Partha Pratim Das
Objectives & Outline
Example - Phone Hierarchy
Summary

SmartPhone ISA MobilePhone

Base Class
class MobilePhone : public LandlinePhone {
protected:
    //PhoneNumber number_1
    //Base subscriber_
    //RingTone rTone_
    AddressBook abook_;
    PhoneNumber *lastDial_1;
    void SetLastDialed(const PhoneNumber& p);
    void ShowDialer();
public:
    MobilePhone(const char *num,
                const char *sub) :
        LandlinePhone(num, sub), // Base ctor
        lastDial_(0) {
        void Call(const PhoneNumber& sp); // Override
        void Call(const Base& s); // Overload
        //void Answer();
        void Redial();
        void SetRingTone(RingTone::RINGTONE r);
        void AddContact(const char *num = 0,
                        const char *sub = 0);
        friend ostream &operator<<(ostream& os,
                                      const MobilePhone& p);
    }
}

Derived Class
class SmartPhone : public MobilePhone {
protected:
    //PhoneNumber number_1
    //Base subscriber_
    //RingTone rTone_;
    AddressBook abook_;
    PhoneNumber *lastDial_1;
    void SetLastDialed(const PhoneNumber& p);
    unsigned int size_1;
    void DisplayPhoto();
public:
    SmartPhone(const char *num,
                const char *sub) :
        MobilePhone(num, sub), // Base ctor
        lastDial_(0) {
        void Call(const PhoneNumber& sp); // Override
        void Call(const Base& s); // Overload
        //void Answer();
        void Redial();
        //void SetRingTone(RingTone::RINGTONE r);
        //void AddContact(const char *num = 0,
                        const char *sub = 0);
        friend ostream &operator<<(ostream& os,
                                      const SmartPhone& p);
    }
}

NPTEL MOOCs Programming in C++
Partha Pratim Das
14
```

We can use that to go further and relate this now with the smart phone. I am trying to model that smart phone ISA mobile phone. So, smart phone is a mobile phone I make the inheritance, these were earlier commented out because they were common from the landline phone. In the smart phone, these members are also not required. These data members and methods and also not required because they will get inherited from the mobile phone, but I need to add the data member and member function that are specific to smart phone.

Coming to these naturally answer gets inherited, but call both of these versions of overloaded call functions in mobile phone need to be again overridden. Now, we can see that this was overridden and this is again being overridden because the way you call through a touchscreen is quite different this is also. So, here in terms of the mobile this was the overloaded in terms of the smart phone. This is now overridden from the definition that you had used in the mobile phone, you are not changing the interface any more, but you will certainly have a different implementation and all of these additional functions are certainly inherited. Of course ReDial() also need to be overridden because if your call is overridden redial is just another version of the call function

(Refer Slide Time: 31:23)

The screenshot shows a presentation slide titled "Phone Hierarchy". The slide contains C++ code illustrating a class hierarchy for phones. The code defines classes for Phone, LandlinePhone, MobilePhone, and SmartPhone, each with its own set of methods and protected members. The code is as follows:

```
class Phone {
public:
    virtual void Call(const PhoneNumber *p) = 0;
    virtual void Answer() = 0;
    virtual void Redial() = 0;
};

class LandlinePhone: public Phone {
protected:
    PhoneNumber number_;
    Name subscriber_;
    RingTone rTone_;
public:
    LandlinePhone(const char *num, const char *sub) : number_(num), subscriber_(sub), rTone_() {
        void Call(const PhoneNumber *p);
        void Answer();
        friend ostream << (ostream& os, const LandlinePhone& p);
    }
};

class MobilePhone : public LandlinePhone {
protected:
    AddressBook abook_;
    PhoneNumber *lastDial_;
    void SetLastDial(const PhoneNumber *p);
    void ShowNumber();
public:
    MobilePhone(const char *num, const char *sub) : LandlinePhone(num, sub), // Base ctor
        lastDial_(0) {
        void Call(const PhoneNumber *p); // Override
        void Call(const Name n); // Overload
        void Redial();
        friend ostream << (ostream& os, const MobilePhone& p);
    }
};

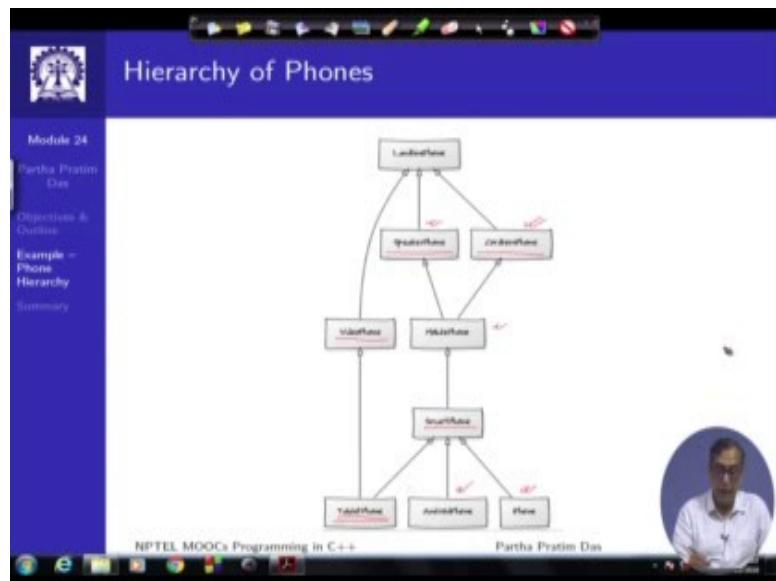
class SmartPhone : public MobilePhone {
protected:
    unsigned int size_;
public:
    SmartPhone(const char *num, const char *sub) : MobilePhone(num, sub), // Base ctor
        lastDial_(0) {
        void Call(const PhoneNumber *p); // Override
        void Call(const Name n); // Overload
        void Redial();
        friend ostream << (ostream& os, const SmartPhone& p);
    }
};
```

The slide also includes a sidebar with navigation links: Module 24, Partha Pratim Das, Objectives & Outline, Example - Phone Hierarchy, and Summary.

So, with this put together, now we have if I combine them that I we have a landline phone, we have a mobile phone which is landline phone specialization. We have a smart phone which is a specialization of the mobile phone and this is how the class interface would look like and what I just outlined here, I will not discuss it here now because we have not at ready with all the C++ features to discuss this is. If we have a hierarchy like this then how about generalizing this further and just looking at a concept of a phone.

So, the basic concept of a phone is I can call, I can answer, and I can redial. So, can I think about an abstract phone, which is the generalization of all kinds of phone. So, I say landline phone ISA phone which is an abstraction of all kinds of phone. Then a mobile phone ISA landline phone and smart phone ISA mobile phone and so on and there is lot of advantages of being able to define such abstraction at the root of your hierarchy which will be the topic of our discussions in the modules may be talk about polymorphism, but this is what shows the you can create complete phone hierarchy and just for your I mean if still, if it looks very straight it is multi level simple hierarchy.

(Refer Slide Time: 32:45)



Then would like to draw your attention to something little bit more realistic in terms of what the phones look like. So, you have a landline phone, which may be a cordless hand set phone or it could be speaker phone. When you have a mobile phone then it is cordless phone as well as a speaker phone because you can use it for both on the other side there are video phones which are landed these days. So, where you could actually make video calls and then you have smart phones as specialization of mobile phones that we have seen, but they are again specialized with whether they are i-phone or an android phone or I could have some kind of a tablet phone which is smart phone as well as can be used as a video phone and so on. So, I just suggest that based on these hierarchy we could try at home and try to built the similar set of C++inheritance classes which can represent this hierarchy

(Refer Slide Time: 33:45)

The image shows a presentation slide titled "Module Summary" for "Module 24". The slide is presented by "Partha Pratim Das". On the left sidebar, there are links for "Objectives & Outcome", "Example - Phone Hierarchy", and "Summary". The main content area contains a bullet point: "Using the Phone Hierarchy as an example analyzed the design process with inheritance". At the bottom of the slide, the text "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" are visible, along with the number "17".

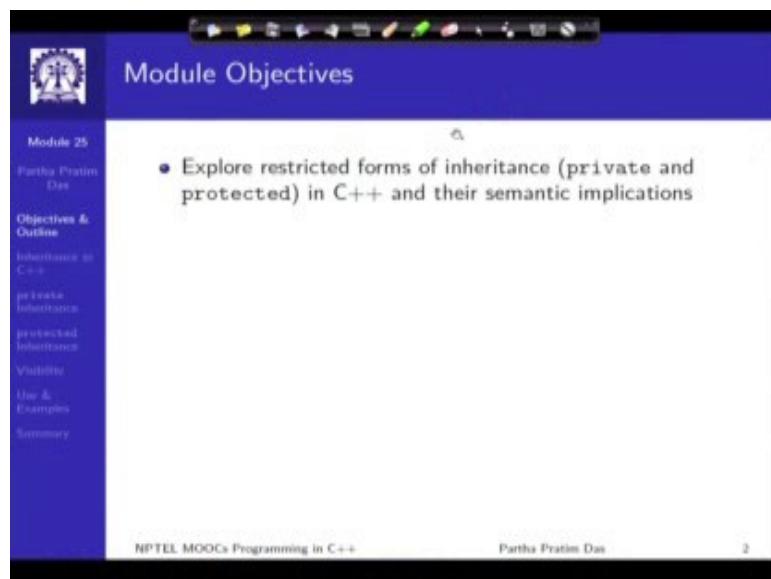
So, to summarize we have used the phone hierarchy here to show, how inheritance can be used to create effective C++ code models for a realistic situations.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 40
Inheritance: Part V

Welcome to Module 25 of programming in C++. We have been discussing Inheritance and with the introduction of all major features of inheritance for generalization specialization or inheritance for ISA hierarchy in C++, we have introduced all the features they are semantics. And in the last module, we took the example of different kinds of phones, a hierarchy of phones; and illustrated how we can start actually designing a set of classes with data members and member functions to realize a hierarchy.

(Refer Slide Time: 01:05)



Module Objectives

- Explore restricted forms of inheritance (private and protected) in C++ and their semantic implications

Module 25
Partha Pratim Das
Objectives & Outline
Inheritance in C++
private inheritance
protected inheritance
Visibility
Use & Examples
Summary

NPTEL MOOCs Programming in C++
Partha Pratim Das

Now in this context, in this basic series of discussions on inheritance, this will be the last module where we explore some more restricted form of inheritance that C++ offered. They distinguish themselves, there are actually of two types private and protected. They distinguish themselves from the inheritance that we have discussed earlier, because they no more they are called inheritance in C++, but they are no more represent the

generalization specialization hierarchy of the object oriented sense.

(Refer Slide Time: 01:41)

This slide shows the module outline for Module 29, titled "Inheritance in C++". The outline includes:

- ISA Relationship
- Inheritance in C++
 - Semantics
 - Data Members and Object Layout
 - Member Functions
 - Overriding
 - Overloading
 - protected Access
 - Constructor & Destructor
 - Object Lifetime
- Example – Phone Hierarchy
- Inheritance in C++ (private)
 - Implemented-As Semantics

The sidebar on the left lists topics such as Objectives & Outline, Inheritance in C++, private inheritance, protected inheritance, Visibility, Use & Examples, and Summary. The footer indicates NPTEL MOOCs Programming in C++ and Partha Pratim Das.

So, they are more used for implementation then for actually modeling or the interface or at the design stage. So, in terms of this outline, we are at the discussion of such restricted inheritance.

(Refer Slide Time: 01:54)

This slide is titled "Inheritance in C++: Semantics". It discusses the Derived ISA Base relationship and provides code examples:

- Derived ISA Base

Diagram illustrating the relationship:

```
graph LR; Base --> Derived
```

Code examples:

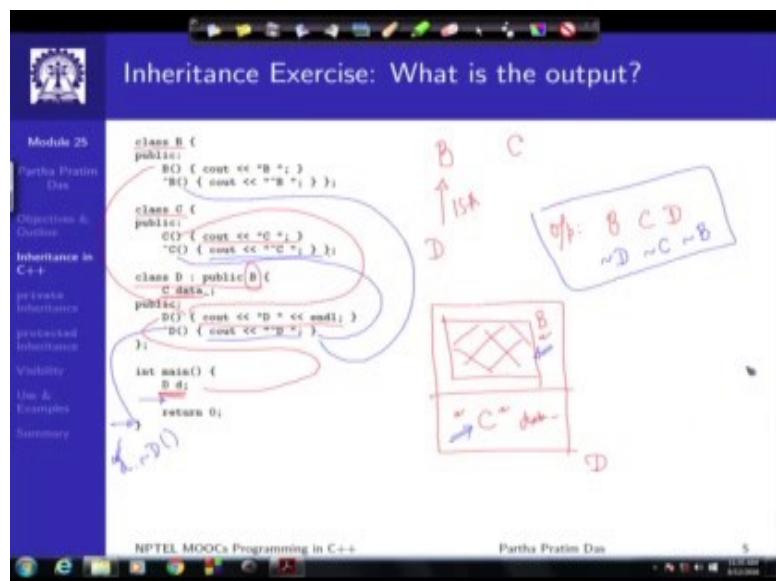
```
class Base; // Base Class = Base
class Derived: public Base; // Derived Class = Derived
```

- Use keyword **public** after class name to denote inheritance
- Name of the Base class follow the keyword

The sidebar on the left lists topics such as Objectives & Outline, Inheritance in C++, private inheritance, protected inheritance, Visibility, Use & Examples, and Summary. The footer indicates NPTEL MOOCs Programming in C++ and Partha Pratim Das.

So, this is the basic inheritance semantics which we have seen and I would like to draw your attention to this public keyword which defines the typical OOAD ISA relationship in terms of an inheritance in C++ classes.

(Refer Slide Time: 02:10)



So, just to recapitulate. If I have a set of classes and we want to know how these classes or objects of these classes will get constructed. So, if you have given such a situation, then you have class B, you have class C, you have class D, which is a specialization of class B, and then you are constructing the object of class D. The question is what is the order in which the cout happen which will tell you what is the life cycle or lifetime of the D object and the other embedded objects in this.

So, whenever you encounter such situations you should quickly try to draw the basic diagram of the relationships. So, here we know that there is a B and C is an independent class. So, this is the basic relationship of the classes. In terms of a D object what do we expect, so this is a D object; in terms of that we have a C data and it is inherited from B. Since it is inherited from B, it will have a base class instance of type B; and as a member, it will have a C class object. So, this is what a basic structure of the layout is. So, once you have been able to draw this, you will be easily able to figure out as to what happens when I try to instantiate an object. Certainly the instantiation of the object, we will start

by invoking the constructor of D.

And then what is the first thing that needs to be done D ISA B, so the base class instance of that will have to be constructed. So, this means that the first thing that this in turn will call is the constructor of B. And once a constructor of B is called it is executed. So, in the output, if I write the output here in the output you see an output of B. Once this base class instance has been constructed, then the next thing is to construct the data members one by one, so there is one data member. Now this will have to get constructed. So, after this D will give the next call to the constructor of C, because this data _ member will have to be constructed. So, as it gives call to the constructor of C will have an output C.

Now once this has been done, so the base part instance has been created, the data member has been created, now the actual constructor of D will execute, so you will have an output D. So, after all these have happened you are now at this position where your whole D object has been constructed. At a later point of time, when you reach the end of the scope, naturally it will have to get destructed. So, if it has to get destructed what will get called first, so this is what you cannot see actually what the compiler has put here invisibly is a call to the destructor of D. So, at this point, you will have a call to the destructor of D. And as I said the destruction order is just the opposite. So, the destruction of D whatever actions are required that will be taken first. So, we will first have the output of ~D. When you reach the end of this, what was constructed just before this was the C the data member. So, this will lead to the call of the destructor of C, which will destroy C will in the output you see this.

And at the end of this scope, it has finished. So, it comes back here again and now you have to destroy the base part instance of the D object. So, you again call the next one which is the destructor of the base class which will print B. And then the function will actually end. So, if you have different hierarchies, it is a good exercise to just put the different print messages in the constructor destructor and trust them to be able to understand how really, what really is a lifetime that happens. So, this was just kind of a recap on what we did.

(Refer Slide Time: 07:13)

Inheritance Exercise: What is the output?

```
Module 25  
Partha Pratim Das  
Objectives & Outline  
Inheritance in C++  
private inheritance  
protected inheritance  
Visibility  
Use & Examples  
Summary
```

```
class B {  
public:  
    ~B() { cout << "B" << endl; }  
};  
  
class C : public B {  
public:  
    ~C() { cout << "C" << endl; }  
};  
  
int main() {  
    B d;  
    return 0;  
}
```

Output:

```
B C D  
~D ~C ~B
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, I just worked out another example, so that you can understand it better. In the next slide, I have actually since I wanted to work out. So, I output was not given in the earlier slide, now it is given here. So, you can practice at a later point of time

(Refer Slide Time: 07:27)

private Inheritance

```
Module 25  
Partha Pratim Das  
Objectives & Outline  
Inheritance in C++  
private inheritance  
protected inheritance  
Visibility  
Use & Examples  
Summary
```

- **private Inheritance**
 - Definition
 - class Base;
 - class Derived: **private** Base;
 - Use keyword **private** after class name
 - Name of the Base class follow the keyword
 - **private** inheritance **does not** mean generalization / specialization
- Private inheritance means nothing during software design, only during software implementation
- Private inheritance means is-implemented-in-terms of. It's usually inferior to composition, but it makes sense when a derived class needs access to protected base class members or needs to redefine inherited virtual functions

- Scott Meyers in Item 32, Effective C++ (3rd. Edition)

NPTEL MOOCs Programming in C++ Partha Pratim Das

Now coming to private the restricted forms of inheritance let me first discuss about

private inheritance. Private inheritance looks pretty much like the inheritance that we have seen which by contrast will say is the public inheritance or just simply inheritance. The only thing different is the key word that we use, you said this is the private inheritance so you write the keyword private. And private inheritance as we understand is not meant for representing the hierarchy in the design, rather it is used for kind of saying that some part of the object is implemented as some other base class object.

So, it is more like a kind of a candidate alternative for composition, which will see some are later in this presentation in this discussion and it is not to be confused with the basic ISA hierarchy that we are modelling to the public inheritance. So, that is why one of the very famous authors' comments that private inheritance means nothing during software design, because during the design time you just know the ISA hierarchy is you do not know the internals of the implementation. So, only when you get to know the internals of the implementation on you are working on that that is where you look into this private inheritance kind of semantics.

(Refer Slide Time: 08:54)

The screenshot shows a presentation slide titled "private Inheritance". The slide is divided into two main sections: "public Inheritance" and "private Inheritance".

public Inheritance:

```
class Person { ... };
class Student : public Person { ... };

// anyone can eat
void eat(const Person& p);
// only students study
void study(const Student& s);

Person p; // p is a Person
Student s; // s is a Student
eat(p); // fine, p is a Person
eat(s); // fine, s is a Student,
// and a Student is-a Person
study(s); // fine
study(p); // error! p isn't a Student
```

private Inheritance:

```
class Person { ... };
class Student : // inheritance-is now private
private: Person { ... };

// anyone can eat
void eat(const Person& p);
// only students study
void study(const Student& s);

Person p; // p is a Person
Student s; // s is a Student
eat(p); // fine, p is a Person
eat(s); // error! s is-a Person
study(s); // error! s isn't a Person
```

At the bottom of the slide, there is a note: "Compilers converts a derived class object (Student) into a base class object (Person) if the inheritance relationship is public." Another note states: "Compilers will not convert a derived class object (Student) into a base class object (Person) if the inheritance relationship is private".

So, if we try to kind of compare between the public and private inheritance, I am just using a simple example here this is again from Scott Meyer's book. I have a base class person, I have a derived class student. And certain I have two different functions eat()

and study(). Certainly eat applies to persons, because everybody needs to eat; and study applies only to students because those who study are called students right. So, there could be persons who do not study. So, if P is a person and S is a student then eat will work on P, interestingly it will also work on S and that is a basic property of public inheritance. We had seen this during the presentation, during the discussion on inheritance itself, I had talked about output streaming operator and showed how passing a derived class object to a base class parameter would still work through the process of casting. So, in terms of the meaning, the semantics what is being said that we expect this to work because someone may be a student, but certainly being a person S also is expected to eat. So, this should work fine.

Similarly, study will work for S, but certainly what will not work is study P, because study is something which is associated with the students. The derived class object and you do not expect the base class the generalized object to be able to satisfy that. So that is what public inheritance mean. Now in contrast in private inheritance, if you look at the same thing the same set of classes, the same functions, the same to P and S, eat P will work, but what is see eat S, if you have defined this as a private inheritance. Now in the context of C++ this is an error, this is the compiler does not allow this. Whereas in case of public it does allow it that is a basic semantic difference.

Why is it not allowed, because here the inheritance. So, it clearly goes to show you that here what we write in the notation of inheritance in C++ as private is not semantically meant to specialize the concept. So, this is just something else, this is just saying that student is possibly implemented as a person and so on. So therefore, the functionality which is for the person is cannot be available to the student, it is component functionality not generalization specialization functionality. So, in terms of private inheritance always keep this concept in mind

(Refer Slide Time: 11:47)

The screenshot shows a presentation slide titled "protected Inheritance". The slide content includes:

- Module 25
- Partha Pratim Das
- Objectives & Outline
- Inheritance in C++
- private inheritance
- protected inheritance
- Visibility Rule & Examples
- Summary

protected Inheritance

- protected Inheritance
 - Definition
 - class Base;
 - class Derived: protected Base;
 - Use keyword **protected** after class name
 - Name of the Base class follow the keyword
 - **protected inheritance does not mean generalization / specialization**
- Private inheritance means something entirely different (from public inheritance), and protected inheritance is something whose meaning eludes me to this day

– Scott Meyers in Item 32, Effective C++ (3rd. Edition)

NPTEL MOOCs Programming in C++ Partha Pratim Das

By symmetry that the third form of inheritance that C++ provide, where instead of writing public or private we could also write protected at the this point of inheritance structure. And we will see the implications of doing that. I would not discuss much of protected inheritance, because it does not offer very distinctive semantics, and many authors have commented that possibly there is no well-defined design situation, where the protected inheritance may be required. So, it is there more for the completeness purpose of the features.

(Refer Slide Time: 12:33)

The slide is titled "Visibility across Access and Inheritance". It features a navigation bar on the left with links to "Module 25", "Partha Pratim Das", "Objectives & Outline", "Inheritance in C++", "private inheritance", "protected inheritance", "Visibility", "Use & Examples", and "Summary".

A "Visibility Matrix" table is shown:

Visibility	Inheritance		
	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	private

Below the matrix is a UML class diagram with three classes: A, B, and D. Class A is the base class, and classes B and D are derived from it. Arrows indicate inheritance: one arrow from A to B is labeled "public", another is labeled "protected", and a third is labeled "private". Class D is also derived from A. A red wavy line is drawn under the "private" inheritance arrow from A to D, indicating that the visibility is private.

Now, what is important is if we have, so now you have two sets of parameters that control how we get the access, how we get the visibility across the base and the derived class. So, the question is if I just draw, I could have base and specialized as D which is basically public, or I could have base and there is no specific notation available. So, I am just using some wavy notation that this is where the derived class D is derived through a protected manner. And similarly, I have base and let say another this notation also does not exist, this is just I am creating it to represent say this means derived in the private way.

Now the question is between these three types of inheritance, public protected and private, in addition to this the base class has three sets of visibility already available. Some members are public, some members are private, and some members are protected. So, in the derived class, what would happen to the visibility of these members and even downwards when I go further below from the derived class what will happen to the visibility of those (Refer Time: 14:17), so that is defined by this visibility matrix. So, here we say what are the visibilities that is, what are the access specifiers given in the base class. And here and the column we say what is the kind of inheritance public, protected or private that is used, and their combination decide as to what is the resultant visibility of an object. So, if I have a publicly visible member, and the derived class

inherits it using public inheritance then the resultant member would remain to be public, but if it inherits it by say the private inheritance, the resultant member will come private.

So, in other words say if some member has a private visibility, private access specifier, inheritance is public, the resultant visibility in the derived class would become private that is in the derive class this will be treated as if it is a private member and so on. So, it is I mean you may feel that you will have to remember these nine entries, but there is very simple thumb rule to understand this. We now understand that public is a most open visibility that is list encapsulated visibility. Protected is little bit more encapsulated than that and private is the most encapsulated visibility. So, there is the hierarchy between these three.

So, if you look into this, basically a hierarchy is considered on this as well. So, if you are trying to understand, what this entry should be, you should look at what are the visible access specifier and what is the inheritance whichever is more restricted out of these two will be the result. So, public private will be private. Similarly if I have private public, I will have private. If I have protected public, I will have protected. If I have protected private, I will have private because between the protected and private, private is more restrictive. So, this basically the thumb rule is simple that you need to be needed to preserve the encapsulation as much as are possible. So, you become more restrictive and this diagram below clearly visually depicts as how the access specification gets re positioned after the inheritance has taken place.

Certainly, before discussing the restricted types of inheritance, we were not we did not have to discuss this, because if you look into just the first column, since the inheritance is the list restrictive inheritance is the provides the list encapsulation, therefore, the visibility of the data members or the member functions in a derived class is simply same as their visibility or visibility in the base class. It is public is public, protect it is protected, private is private. But as we introduce other forms of inheritance, this as access restriction description is becomes required.

(Refer Slide Time: 17:37)

The slide title is "Inheritance Exercise: What is the output?". The slide content includes the following C++ code:

```
Module 25
Partha Pratim Das
Objectives & Outcome
Inheritance in C++
private inheritance
protected inheritance
Visibility
Use & Examples
Summary

class B {
protected:
    BO { cout << "B "; }
    "BO { cout << "B "; }
};

class C : public B {
protected:
    CO { cout << "C "; }
    "CO { cout << "C "; }
};

class D : private C {
public:
    DO { cout << "D " << endl; }
    "DO { cout << "D " }

int main() {
    D d;
    return 0;
}
```

Below the code, there is a hand-drawn diagram illustrating object layout. It shows four boxes labeled B, C, D, and D. Box B has an arrow pointing to it from the text "B". Box C has an arrow pointing to it from the text "C". Box D has two arrows pointing to it from the text "D". Below the boxes, the letters B, C, B, C, D are written in red, with a red arrow pointing to the last 'D'.

So, with this, we can again go back and try to look at some of the little bit exercise. This is an exercise showing you the similar structure as before, but we have here both public and private inheritance. So, if we want to see again in terms of the construction of D what should happen? Then let me, this is the class B we know, C ISA B so this is what we draw here. Then we have we say that D ISA private C. So, let us just draw it like this. So, this is what we get to see. Then in terms of the object layout we find the D object should have a base part, which is C, the base part is C. Then it has a data part, which is also C, this base part is C as well as the data part is also C, and that is all that we have. We have C is a B, C is a B here, so C again will have a base part which is B and this will have a base part, which is B. We have every C object will have a base class instantiation because is a specialization here.

So, given this, if you now try to trace, how the objects will constructed, certainly as soon as this is encountered the constructor of D is called, so the base has to get instantiated. So, the constructor of C will get called. So, the base of that will have to get constructed. So, first thing that will get constructed is the B object, this B object. Then certainly the construction of the base object completes. So, C will get to that completes the construction of this. Then that data member this data member has to get constructed which also has base part so that will get to constructed then the data member gets

constructed, and then finally the D object gets constructed. So, this is the order in which the construction will happen.

So, in terms of private inheritance also the life time issues do not get affected, life time issues remain same. So, we had seen if this kind of an exercising terms of just public inheritance. So, intentionally I have introduce private inheritance here that changes the access specification, what you can access, what you cannot access, but in terms of the dynamics of the object life time things do not really become different, they remain to be same. The full solution is given in the next slide here.

(Refer Slide Time: 20:50)

The screenshot shows a presentation slide titled "Inheritance Exercise: What is the output?". The slide content includes:

```
Module 29
Partha Pratim Das
Objectives & Outline
Inheritance in C++:-
private inheritance
protected inheritance
Visibility
Use & Examples
Summary
```

```
class B {
protected:
    B() { cout << "B" << endl; }
};
class C : public B {
protected:
    C() { cout << "C" << endl; }
};
class D : private C {
    C data_;
public:
    D() { cout << "D" << endl; }
    ~D() { cout << "D" << endl; }
};

int main() {
    D d;
    return 0;
}
```

Output:

```
B
C
B
C
D
~C
~B
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, at leisure, you can work out and get convenience that this is what is going to happen.

(Refer Slide Time: 21:01)

The slide is titled "Inheritance Exercise: Access Rights". It features a class hierarchy diagram and a code snippet. The code defines classes A through G with their access specifications:

```
class A {
    private: int x,y;
    protected: int y;
    public: int z;
};

class B : public A {
    private: int u;
    protected: int v;
    public: int w; void f() {x;}
};

class C : protected A {
    private: int u;
    protected: int v;
    public: int w; void f() {x;}
};

class D : public B {
    private: int u;
    protected: int v;
    public: int w; void f() {x;u;}
};

class E : public B {
    public: void f() {x;u;v;w;}
};

class F : public C {
    public: void f() {x;u;v;w;}
};

class G : public D {
    public: void f() {x;y;z;u;v;w;}
};
```

Annotations on the slide include:

- Inaccessible Members:** Handwritten annotations highlight members that cannot be accessed from outside their respective classes.
- Accessible Members:** Handwritten annotations highlight members that can be accessed from outside their respective classes.
- Diagram Labels:** Labels A through G are placed near the corresponding classes in the hierarchy diagram.
- Handwritten Notes:** There are several handwritten notes, including "private" and "public" labels, and arrows indicating access paths between classes.

To understand the access specification, I have included an example again I will just do couple of steps and then leave it as an exercise for you to work out just first analyze what is there class A, so there is a class A. So, whenever you have this hierarchy related designs or hierarchy related issues, it Is good to just quickly draw a diagram, so that mentally you can see what is really going on. It is more difficult to frequently refer to I am sorry frequently refer to code and talk about this. So, we have B ISA A, then we have this. So, we say C ISA A; we also have D ISA A. This is public type one, this is protected, and this is private. And then we have these are E ISA B public; F ISA C, which is public; G ISA D which is public. So, this is the basic hierarchy that we have.

So, if you have that then, and we have different members here x, y, z - three members with three kinds of visibility; private, protected and public. And B adds u, v, and w, so does C, so does D, and these data members are also of private, protected and public kind. So, having done that our question is if we now look into the different access of variables in different member functions and this is a global function then what members can be access what members cannot be accessed. So, the way I demonstrate here is, here I only show I have shown on the left side, I have only shown the inaccessible members that is if I have a class f in I am sorry a function f() in class B, then x is in accessible to this function. Why is it so? Where is the x coming from, B does not have x, but B ISA A has

an x. So, this x actually means that it is A: :x that has been inherited. And what is the access specifier private. So, certainly the derived class cannot access the private data members of the base class. So, this is inaccessible.

If you look at x in class C, it is again the same logic that this is private in A, therefore, you cannot access it; you cannot access it either in class D, because it is private in A, clear. Now let us say if we look into the class E, and the function f() that the class E defines the fact that x will be inaccessible is clear, because x is private here therefore, x is private in B, x is private in C, x is private in D. Because it is private for B, it is private specifier public inheritance, resultant visibility private. For C, it is public sorry private specifier protected inheritance, protected inheritance, resultant is private. In A, this is specifier is private inheritance is private resultant in D, this is private. So, in all of B, C, D, we can reason that x is actually private. So, this access will not be accessible to any of this.

But in E, you find that u is not accessible. Why u is not accessible, what is u you are sitting here. So, where can it get in u from it, it does not have any data member itself. So, it can get the u from the parent class B. So, it B has a u. So, this u is basically B : : u which is private member of B therefore, you cannot access it here. You can do similar reasoning for the class F. Let us look at class G, class G cannot access, so many different things for example, class G cannot access y, class G cannot access y. So, where does it get y form, it does not have a y itself. So, where does it I have to get it, it will have to get on this path. So, let us it is parent is D, D does not have any y; u, v and w, it is does not have any y. So, where can it get if from, from the parent A has a y.

So, you know here, this is protected, this y is protected. So, the child class should be able to access it that is D should be able to access it. Can D access it, if I go to D, if I go to D, the function in D this f function can access y and that is a reason it is not mentioned here that is quiet Ok because you can always access it from the parent, but what happens D has inherited a through a private inheritance mechanism. So, the y which was protected in A now becomes private in D, it is not written, but it is in within the inheritance this has become a private member and certainly no derived class can access a private member, so g cannot access is that is a reason you will not be able to g will not be able to access y.

You can reason through and see the same similar faith for these variables as well.

Going further, you could do, I will not do this now, you can do, you should be able to work this out. This is where we saw over this hierarchy, how does the access of different members actually again I have shown it all for the data members, but the similar reason you will apply for the member functions as well, how will this different forms of inheritance interact with the difference specifier will is clear in terms of the member functions. You could just now again try to solve this for a global function. And all that you will have to follow is a similar reasoning of going over the hierarchy. Wherever if you are trying to say that c.w is accessible then c is of class C, so it is here. So, you see does C have a w C has a w. And being a global function it has to be able it will be able to access only those members which are public. So, C has a w which is public therefore, this is accessible.

But you think about any of the other variables that any of the other members that is C has none of them are accessible to for example, v is not accessible, because v here is protected; z is not accessible, why z not accessible, because if I talk about c.z, then there is no z here. So, the z comes from A, which is public, it should have been accessible, but what has happened in the process of inheritance you have inherited it as a protected inheritance; C is inheriting A through the protected manner. So, the resultant visibility of z has become the lesser of public and protector. So, this is become protected member and the protected member is not visible outside the class, since F is a global function here it is not visible to F. So, c.z is not accessible. So, in this way, I would request that you work out for this whole set and that will clarify all your doubts that you may have.

(Refer Slide Time: 30:02)

```
Simple Composition
#include <iostream>
using namespace std;

class Engine {
public:
    Engine(int numCylinders) { }
    // Starts this Engine
    void start() { }
};

class Car {
public:
    // Initializes this Car with 8 cylinders
    Car() : e_(8) { }

    // Start this Car by starting its Engine
    void start() { e_.start(); }

private:
    Engine e_; // Car has-a Engine
};

int main() {
    Car c;
    c.start();
    return 0;
}

private inheritance
#include <iostream>
using namespace std;

class Engine {
public:
    Engine(int numCylinders) { }
    // Starts this Engine
    void start() { }
};

class Car : private Engine { // Car has-a Engine
public:
    // Initializes this Car with 8 cylinders
    Car() : Engine(8) { }

    // Start this Car by starting its Engine
    using Engine::start;
};

int main() {
    Car c;
    c.start();
    return 0;
}
```

Now, finally, let me just try to show you that some possibilities of using private inheritance. We take one example again you should look at the left hand side first where I have an Engine class, and have a Car class, and the basic concept is that the car has a Engine. So, if the car has a engine it has component. So, I can model it in this manner. So, that the constructor of the Car instantiates the Engine which invokes the constructor with 8 cylinder value. So, I have a Engine instance here. And therefore, and in terms of starting that Engine, Car has provided method start() which actually in turn invokes the method start() of the engine. And if I do this, the car will get started, simple composition based model.

(Refer Slide Time: 31:03)

The slide title is "Car HAS-A Engine: Composition OR private Inheritance?". It features two code snippets side-by-side:

Simple Composition:

```
#include <iostream>
using namespace std;

class Engine {
public:
    Engine(int numCylinders) { }
    // Starts this Engine
    void start() { }
};

class Car {
public:
    // Initializes this Car with 8 cylinders
    Car() : e_(8) { }

    // Start this Car by starting its Engine
    void start() { e_.start(); }

private:
    Engine e_; // Car has-a Engine
};

int main() {
    Car c;
    c.start();
    return 0;
}
```

private inheritance:

```
#include <iostream>
using namespace std;

class Engine {
public:
    Engine(int numCylinders) { }
    // Starts this Engine
    void start() { }
};

class Car : private Engine { // Car has-a Engine
public:
    // Initializes this Car with 8 cylinders
    Car() : Engine(8) { }

    // Start this Car by starting its Engine
    void start() { Engine::start(); }
};

int main() {
    Car c;
    c.start();
    return 0;
}
```

Annotations in red highlight the differences between the two approaches. Handwritten notes include "Simple Composition" next to the first snippet and "private inheritance" next to the second snippet. There are also small drawings of a car and an engine.

Now I show in parallel for comparison that this could also be modeled in terms of private inheritance, I have the same engine class no difference, but what I change is I do not use this private data member instead I inherit from the Engine class in a private manner. If I inherit from the Engine class is a private manner, what do I have, here I had there are the class and the Car class and I had the Engine as a data member. Here since I have inherited I will still have the Engine as a base class instance, I have the Car here and this is a base class instance, so that that way I will be able to have the value, but certainly what I need to do I need to have ability to start the engine. So, that is what I specify here this is just another notation where you using. So, when you say using base class :: method name then it means that whenever I talk of this name, it actually means the base class item, it means that base class function. So, if I do this then I can again instantiate the car and do c.start; and c.start through this using will actually call the start of the engine, which is again the engine will get started.

So, in here also the engine is not visible to others, because is a private data member, here also this is not accessible to others, because I have done a private inheritance. So, this is another way that some implementation can be I mean you could implement some kind of a component as private inheritance, and certainly this brings us to question of which one should we use naturally, there is no question of using private inheritance in a usual

situation, you should always use composition.

(Refer Slide Time: 33:01)

The screenshot shows a presentation slide titled "private Inheritance". The slide content is as follows:

- Use composition when you can, private inheritance when you have to
- **Private inheritance means nothing during software design, only during software implementation**
- **Private inheritance means is-implemented-in-terms of. It's usually inferior to composition, but it makes sense when a derived class needs access to protected base class members or needs to redefine inherited virtual functions**

— Scott Meyers in Item 32, Effective C++ (3rd. Edition)

The sidebar on the left lists topics: Module 29, Partha Pratim Das, Objectives & Outline, Inheritance in C++, private inheritance, protected inheritance, Visibility, Use & Examples, and Summary. The footer includes NPTEL MOOCs Programming in C++ and Partha Pratim Das.

Only very few special situations where composition does not really work you must use private inheritance. And you use them only when those situations are faced. So, while we have while we discuss polymorphism later in this series, then at a suitable point I will show you examples using polymorphism of certain problems, where the private inheritance can give you actually a better solution than using composition. But as a thumb rule I would say rather 99.99 percent of cases, if there is a situation of composition, you should actually use data members and not use private inheritance in that place.

(Refer Slide Time: 33:46)

The slide is titled "Module Summary" and is part of "Module 25". It features a sidebar with a logo and navigation links: "Partha Pratim Das", "Objectives & Outline", "Inheritance in C++", "private inheritance", "protected inheritance", "Visibility", "Use & Examples", and "Summary". The main content area contains two bullet points:

- Introduced restricted forms of inheritance and protected specifier
- Discussed how private inheritance is used for *Implemented-As Semantics*

At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" with slide number "16".

To summarize, we have introduced here the basic notions of restrictions on inheritance in terms of private and protected inheritance. And we have discussed how really the protocol is defined through which the three kinds of public, protected and private visibility interact with three kinds of inheritance public, protected and private inheritance. And at the end, we have tried to show an example illustrating that how private inheritance can be used to realize what is more commonly known as something is implemented as something else or imp or uses a component of something else and, but with the caution that we will use it in a very restrictive manner.

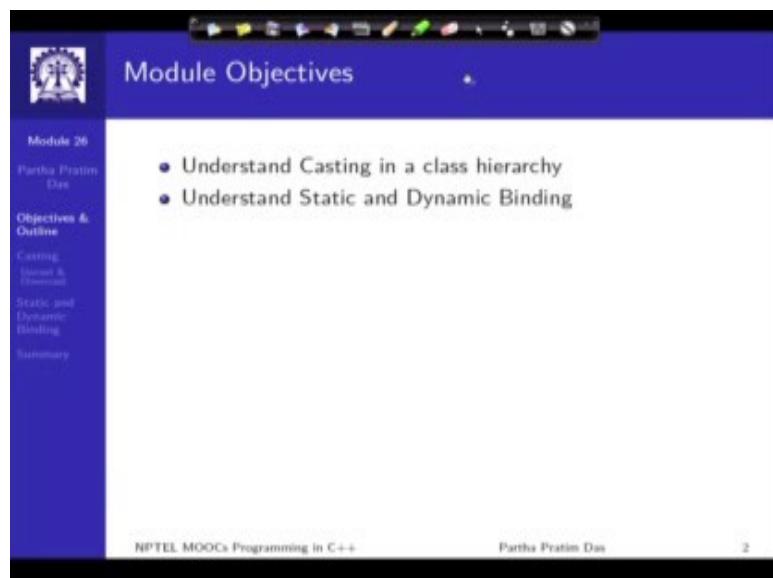
So, with this, we will bring close to the series of basic notions of inheritance. And from the next module onwards, we would talk about advanced aspects of inheritance, which lead to what is known as polymorphism that will be the core idea to discuss.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 41
Dynamic Binding: Part I

Welcome to module 26 of Programming in C++.

(Refer Slide Time: 00:21)



Module Objectives

- Understand Casting in a class hierarchy
- Understand Static and Dynamic Binding

Module 26
Partha Pratim Das
Objectives & Outline
Casting
Static and Dynamic Binding
Summary

NPTEL MOOCs Programming in C++ Partha Pratim Das 2

We have seen the inheritance structure in C++. We will next look into one of the or possibly the most powerful feature of C++ to support object-oriented implementation which is commonly known as the name of polymorphism. But before we can go in depth in terms of the discussion of polymorphism, we need to prepare ourselves with some more understanding in terms of the casting that can happen in a class hierarchy. And the core differences between the static and dynamic binding. So, this dynamic binding discussion will cover couple of modules and this is the first one and will lead us to the total understanding of the polymorphism in C++.

(Refer Slide Time: 01:32)

Module Outline

- Casting
 - Upcast & Downcast
 - Static and Dynamic Binding

Module 26
Partha Pratim Das
Objectives & Outline
Casting (Upcast & Downcast)
Static and Dynamic Binding
Summary

NPTEL MOOCs Programming in C++ Partha Pratim Das 3

So, the outline is casting and static and dynamic binding it will available on the left of your every screen.

(Refer Slide Time: 01:42)

Casting: Basic Rules

- Casting is performed when a value (variable) of one type is used in place of some other type

```
int i = 3;  
double d = 2.5;  
  
double result = d / i; // i is cast to double and used
```
- Casting can be implicit or explicit

```
int i = 3;  
double d = 2.5;  
  
double *p = &d;  
  
d = i; // implicit  
  
i = d; // implicit -- // warning C4244: 'int' : conversion from 'double' to 'int',  
// possible loss of data  
i = (int)d; // explicit  
  
i = p; // error C2440: 'int' : cannot convert from 'double *' to 'int'  
i = (int)p; // explicit
```

Module 26
Partha Pratim Das
Objectives & Outline
Casting (Upcast & Downcast)
Static and Dynamic Binding
Summary

NPTEL MOOCs Programming in C++ Partha Pratim Das 4

So, let us talk about casting. We all know casting, you all know C. So, casting you know is basically performed when a value of one type, certain type is used in place some other

type is used in the context of some other type. So, for example, here *i* is an integer value variable, *d* is a double variable and I am writing *d / i*. So, if I say *d* then if I say */* then the meaning of this */* is a division of floating point numbers. So, what is expected here is also a double value but what I have provided, I have provided an integer value. So, this still works in C how does it work? So there will have to be some process by which this integer value is kind of converted to a double value and then used. So, this mechanism is the mechanism of casting that *i* is cast to double and then it is used. So, this is a simplest notion that we have seen and in C this is particularly called particularly referred to as a mixed mode operation which we all know.

Now, let us go for the start developing little bit more on it we know that the casting can be implicit or it could be explicit. For example, look at first look at the types of the variables *i* is an int variable, *d* is a double variable and *p* is a pointer variable pointer to double. So, if I make an assignment of *i* to *d* then, this is double and this is int. So, certainly they are not of the same type assignment should not be possible, but implicitly it will cast the integer to the double and allow me to do this. I can have the reverse the double being assigned to *i* this will also be allowed, but with a warning for the simple reason because double is a much bigger data type, whereas int is a smaller data type so, some data will be lost.

In contrast, I could use what is called the C style casting that is put the required type with in parenthesis before the available value and use that, this is known as explicit casting. And now the compiler will not shout. So, you can see that if I try do something like *p* is a pointer and *i* is an integer, if I want to assign the *p* to *i* is an error your certainly you cannot take a double pointer and use it as an int. But if I explicitly cast then the C compiler will allow that. These are basic rule of casting.

(Refer Slide Time: 04:53)

The slide has a blue header with the title 'Casting: Basic Rules'. On the left, there's a sidebar with a logo, the text 'Module 26', 'Partha Pratim Das', 'Objectives & Outcome', 'Casting (Implicit & Explicit)', 'Static and Dynamic Binding', and 'Summary'. A small video thumbnail of a man speaking is also on the left. The main content area contains a bulleted list and several code snippets. The bullet point says: '• (Implicit) Casting between unrelated classes is not permitted'. Below it is a block of C++ code showing various assignments between objects of class A and B, each followed by a comment indicating a compilation error:

```
class A { int i; };
class B { double d; };

A a;
B b;

A *p = &a;
B *q = &b;

a = b;      // error C2679: binary '=' : no operator found
           // which takes a right-hand operand of type 'main::B'

a = (A)b; // error C2440: "type cast" : cannot convert from 'main::B' to 'main::A'

b = a;      // error C2679: binary '=' : no operator found
           // which takes a right-hand operand of type 'main::A'

b = (B)a; // error C2440: "type cast" : cannot convert from 'main::A' to 'main::B'

p = q;      // error C2440: "=>" : cannot convert from 'main::B *' to 'main::A *'

q = p;      // error C2440: "=>" : cannot convert from 'main::A *' to 'main::B *'

p = (A*)b; // Forced -- Okay
q = (B*)a; // Forced -- Okay
```

At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

Let us move on let say, if we try to now extend this to C++. So, earlier we were talking about built in types only int, pointer, double and so on. Now, I have two classes A and B and I have two objects, I have two pointers one of A type another is of B type which will holds the address of A, and the address of B. So, if I try to make this assignment this assignment is a casting error because what is this assignment saying? This is saying that a assigned to b means a.; we all know it now operator assignment b. So, it means that it the class A to which this a belongs, this a belongs to class A must have an operator ‘=’ which takes an operand of type b which does not exist.

If we try to explicitly cast, this will also fail because there is no explicit cast defined. Is not like int, double where you know how to convert an int to a double or double to a int. So, if you try to do them in the reverse way of course, this will fail. If you try to cast the two types of pointers pointer to a into pointer two b this will fail, the other direction this will also fail. So, all of these are casting failure because you are not allowed to do this. The only thing that you can force is you could take the & b , & b is of type B* and force it to be A*. That is you can convert the pointer of unrelated classes one into to the other, but in a forced manner. Naturally, you could not do this p = q is not allowed. This of B*, this is of A*. You cannot make an assignment, you cannot use. But if you forcefully say that the B* is converted to A* a pointer, then you can make this assignment. The

symmetric one will also work. So, this is basically that the story of casting between classes when we introduce C++.

(Refer Slide Time: 07:11)

The screenshot shows a presentation slide titled "Casting: Basic Rules". The slide is part of "Module 26" by Partha Pratim Das. On the left, there's a sidebar with navigation links: "Objectives & Outline", "Casting", "Unions & Unions", "Static and Dynamic Binding", and "Summary". A small video thumbnail of the speaker is also present. The main content area contains a bullet point: "Forced Casting between unrelated classes is dangerous". Below it is a code snippet:

```
class A { public: int i; };
class B { public: double d; };

A a;
B b;

a.i = 5;
b.d = 7.2;

A *p = &a;
B *q = &b;

cout << p->i << endl; // prints 5
cout << q->d << endl; // prints 7.2

p = (A*)&b;
q = (B*)&a;

cout << p->i << endl; // prints -868900459 ----- GARBAGE
cout << q->d << endl; // prints -9.25696e+061 ----- GARBAGE
```

Now, if we do forced casting as we just saw between unrelated classes then, the results may be dangerous results may be actually very unpredictable. So, I have just the similar two classes. Now, I have just put in the data members in public. I want to just demonstrate you. So, these data members have been initialized and I have a pointer of type p which keeps a type A which keeps the A address the q and I print the values certainly the values are correctly printed. $a \rightarrow i$ is basically a.i which is 5, $q \rightarrow d$ is 7.2.

Now, suppose I have forcefully cast it. So, if I have forcefully cast it as said that this is what I want. Is this is a b object and the pointer to that I have cast as a pointer and then I am trying to do this. This is what it prints. If you try this it is not necessary that it will print this. In my system it printed this, but it will certainly print some garbage. Why is this happening? This is happening because if you look into the a object this as a int. let us say we are on a 32 bit system. So, int has 4 bytes in which this value 5 is written. In a b object I have a double which has possibly 8 bytes. Now, if I take the pointer p which is of type A; that means, the pointer if I write $p \rightarrow i$ then it the compiler always know that it

should point 2 and a type of object so that it can take 4 consecutive bytes and make it an integer.

Now, that is what I have violated. What I have done? I have actually taken the p and make it point to the b address. Now here what was written here in 8 bytes the 7.2 was written, but p knows that it is of type a. So, it knows that it has to read 4 bytes and think it is an integer. So, it takes that floating point representation of 7.2 arbitrary reads the possibly the first four bytes and start thinking that it is an integer and prints it as an integer.

When I do the reverse think of when I do the reverse then what I am doing? I have q pointer which is of pointing to b type. So, if I do $q \rightarrow d$ it expects 8 bytes to be representing a double. Now, I have made q point to this object a, which means that it is actually reading 8 bytes of which the first 4 bytes is a integer representation of 5. The next 4 byte, God knows what is there. It is invalid part of memory, it goes there takes that value interpret it is at a floating number and prints. So, that is the basic problem that casting can get you to. So, force casting is a dangerous thing.

(Refer Slide Time: 10:48)

The screenshot shows a presentation slide with a blue header containing the title 'Casting on a Hierarchy'. The slide content includes a bullet point stating 'Casting on a hierarchy is permitted in a limited sense' followed by several code snippets illustrating different casting operations. A sidebar on the left lists course modules such as 'Module 26', 'Partha Pratim Das', 'Objectives & Outcome', 'Casting: Upcast & Downcast', 'Static and Dynamic Binding', and 'Summary'. A small circular portrait of the speaker, Partha Pratim Das, is visible in the bottom left corner of the slide area.

```
● Casting on a hierarchy is permitted in a limited sense
class A () {
    class B : public A () {
        A *pa = 0;
        B *pb = 0;
        void *pv = 0;
    }
}

pa = pb; // okay ----- // UPCAST
pb = pa; // error C2440: "=>" : cannot convert from 'A *' to 'B *' // DOWNCAST
pv = pa; // okay ----- // Lose the type
pv = pb; // okay ----- // Lose the type
pv = pc; // okay ----- // Lose the type

pv = pa; // error C2440: "=>" : cannot convert from 'void *' to 'A *'
pv = pb; // error C2440: "=>" : cannot convert from 'void *' to 'B *'
pv = pc; // error C2440: "=>" : cannot convert from 'void *' to 'C *'
```

Now, let us see how casting will look like if you try to do it on a hierarchy. So, we have class A and class B is a specialization of A. So, I have two pointers of two class types. Now earlier when the types were unrelated it was not possible to assign the pointer of one type in to the pointer of other, but now I can assign pb to pa, and this will be permitted. But if I try to do the reverse assign pa to pb this will not be permitted. The reason is simple. If I have an A object and think about a B object what does B object have? It is a specialization of A. So, B object internally has an A object which is the base object.

So, if I have pa, which is a pointer to A type, it is a pointer to A type. Then, when I assign pb to it, what was pb? pb is a pointer to the B type it is pointer to this. So, when I take this value and put it to A then pa refers to the same area. But, what this pa knows, pa knows that it is a pointer of A type which means pa knows that it will get an A object. So, what will it do? It will simply not take will not be able to take cognizance of the extended part of B, but it will get valid A object which is the base part of the B object and will just give you that object. So, that is quite to do that. You just get a limited view of that.

But if you try to do the other way, if we try to take pb and make it point to the A object then pb things knows that there is a A object as a base part and then there are more things. So, it will try to think that this whole thing is the B object which actually it is not exist only the A object exist. So, this would be something dangerous it will get into violations in terms of the memory. So, that is the reason this is allowed. Let me change the color whereas this is not allowed. And when we are doing this if I go back to the hierarchy diagram then this is the diagram B ISA A. So, as I do this I am moving from the direction of B or specialized object to the generalized object so I am going up. So, we say this is upcast which is allowed. But if I try to move down that is if I take a generalized object and think that it is a specialized one then say downcast which will have to be forbidden.

Of course, in the same example I have also shown that what will happen if you have a void pointer. Certainly you can take objects of any type and yes keep that address in the void pointer all that happens is you lose the type. Similarly, you could do the reverse. Try

to do the reverse that is take a I think there is a typo here what these are typo I will just correct that later on in the presentation. So, this is what is meant is if pa is assigned pv or pb is assigned pv then these all will be error because pa is a void* pv is a void* pointer. So, it does not know, where does it point to? It does not know how many fields what size it should point to. So, if you take that and try to interpret as an A, then certainly you have all kinds of dangers looming. So, these will not be allowed, but it is quite ok if you actually take a pointer of any type and put it to a void pointer. Of course, in C++ we will see that there will be badly any use for that.

(Refer Slide Time: 14:58)

The screenshot shows a presentation slide titled "Casting on a Hierarchy". The slide has a blue header with the title and a sidebar on the left containing navigation links and a photo of a man. The main content area contains code demonstrating up-casting:

```
● Up-Casting is safe
class A { public: int dataA_; };
class B : public A { public: int dataB_; };

A a;
B b;

a.dataA_ = 2;
b.dataA_ = 3;
b.dataB_ = 5;

A *pa = &a;
B *pb = &b;

cout << pa->dataA_ << endl; // prints 2
cout << pb->dataA_ << " " << pb->dataB_ << endl; // prints 3 5

pa = &b;

cout << pa->dataA_ << endl; // prints 3
// cout << pa->dataB_ << endl; // error C2039: 'dataB_': is not a member of 'A'
```

Now, if we look into the upcasting we can we are just enhancing the class with data members and if we look into that if we put a value 2 to the a object, that is the data part of A object value 3 and 5. So, this is a object which is 2. This is the b object which is 3 in the, a part and 5. And then if we take their addresses and try to print the fields. We get the 2 and 3, 5. This is printing the a object, this is printing the b object. Now, let say we have done an upcast. This is a situation of up cast I have taken the address of b and put it in a pointer of type A. So, what does pa get to see? pa is pointing here, but it has the knowledge of only a. So, it gets to see only this part. So, what happens is if I try to print `pa → dataA_` it prints this and you get a 3 as expected, but if you try to do this `pa → dataB_` that is if you try to print this certainly the compiler gives you an error because the

compiler knows that pa is a pointer of A type which does not have any dataB_ member and therefore, this will not be allowed.

So, if you up cast there is no situation where you can get into an error situation because either you will be accessing the base part of the class which is just making limited access which is fine or you and the compiler will restrict you from using or accessing any part of the specialized class which actually does not exist. So, upcasting is safe. Downcast you can argue very easily that downcasting will be risky.

(Refer Slide Time: 17:03)

The screenshot shows a presentation slide with the title "Static and Dynamic Binding" at the top. Below the title, there is a sidebar with navigation links: "Module 26", "Partha Pratim Das", "Objectives & Outline", "Casting", "Virtual & Overload", and "Static and Dynamic Binding Summary". The main content area contains the following C++ code:

```
#include <iostream>
using namespace std;

class B {
public:
    void f() { cout << "B:f()" << endl; }
    virtual void g() { cout << "B:g()" << endl; }
};

class D: public B {
public:
    void f() { cout << "D:f()" << endl; }
    virtual void g() { cout << "D:g()" << endl; }
};

int main() {
    B b;
    D d;

    B *pb = &b;
    B *pd = &d; // UPCAST

    D *pd = b; // DYNAMIC CAST
    D *pd = d; // UPCAST

    pb->f(); // B:f() -- Static Binding
    pb->g(); // B:g() -- Dynamic Binding
    pd->f(); // B:f() -- Static Binding
    pd->g(); // B:g() -- Dynamic Binding

    rd.f(); // B:f() -- Static Binding
    rd.g(); // B:g() -- Dynamic Binding
    rd.f(); // B:f() -- Static Binding
    rd.g(); // B:g() -- Dynamic Binding

    return 0;
}
```

At the bottom left, it says "NPTEL MOOCs Programming in C++". At the bottom right, it says "Partha Pratim Das".

Given this let me just introduce the basic concept of static and dynamic binding and it will become clear slowly as to why did I discuss about the casting before introducing this? So, I have a very simple situation. Let me get back to red. So, a very simple situation there is a base class and there is a specialized derived class B, class D. We have seen this earlier. So, base class has a function f() method f() derived class has inherited and then overridden this method. There is nothing special about that. Base class has another function g() and derived class has overridden that function as well. Only thing different in this is that in case of the function g we have written another additional word keyword virtual and we will see how that changes the behavior. So, this is the situation always keep that diagram in mind.

So, I have created two instances and we have two pointers pb and pd both of the base class type. So, I keep the address of b in the pb pointer which is normal and I keep the address of d in the pointer pd of the B class type which means I am doing an upcast. There is an object here pointer is of this type. So, I am making a representation assignment here. So, I have done upcast. Similarly, the same thing I have written in terms of the reference. This is the reference to the rb is the reference to the b object, rd is a reference to the d object, but the only difference is rd is of type B type reference. So, it will rd will up cast and think of d as if it were a B type object.

In this so this is the setting and in this we are trying to make function calls of all types. So, we are calling for both objects b and d. We will call both functions. So, four combinations and we will call them in three different ways. The first we call the functions using the object. So, b . f(), b . g(), d . f(), d . g(). We know what function will get called b .f(); this function will get called b . g(), this function will get called b .d(), certainly this function will get called. The corresponding function of that class will get called. So, there is no surprise in that.

Now, let us try to make that function calls through pointer. So, pb is a pointer to the b object, pd is a pointer to the d object, but both of them are of B type. So, since they are of B type so if I invoke pb → f() certainly I expect the function of the B class to be invoked. So, pb → f() should invoke B :: f() this function. Similarly the next pb → g() should invoke the g function; pd → f() pd also is of type the base class. So, pd also knows only about these two functions. So, if I do pd → f() it again invokes the function f of the B class. But something totally surprising happens when you do pd → g(). Mind you pd is a pointer of this type. It has it is actually currently pointing to an object of this type. So, the type of pd is here. That object actually is here, but when I invoke the method, it somehow is able to find out that it actually has a d object and instead of invoking this function it actually invokes this function. And that is what is known as dynamic binding.

So, what is binding? Binding is given an expression in terms of invocation you decide which function will get called. So, we have talked about binding in the context of overloading also that if there are multiple functions overloaded which particular function will be bound. So, it is a similar concept. So, given this p pointer pb → f() or pd → g() the

question that we want to ask as to which particular function will get bound by with this call.

Now, what we see is, in case of the function f the binding is static which means the binding is decided by let me again clear it. So, in case of function f here and here the binding is static which means which function it calls depends on the type of the pointer. Which means something that is statically known that is known at the compile time compiler always know what is the type of the pointer, but the fact is this pointer is having a B object pointing to a B object whereas this pointer is pointing to a D object.

In case of static binding those that is no consideration this is a B type pointer this also is a B type pointer. So, if I invoke f this invokes B :: f() that is a f method of the base class. This also invokes f method of the base class. Scenario change in the other case where we say we are having dynamic bind if you look into these now again these two are both are pointers of the base type. And this is pointing to the b object, this is pointing to a d object.

Now, we find that when this pointer is pointing to a b object, B :: g() that is here this function is getting invoked. Where as in a different identical expression where the pointer still is of B* type, but when it invokes g() given that the object it actually is pointing to is a d type object now the D :: g() that is this overridden function in the derived class gets invoked. So, it does not depend on the type of the pointer. So, the compiler has no way to decide as to pb → g() or pd → g() which function will it call because it is not getting decided by the type of the pointer, but it is getting decided by actually at the run time. What is the type of the object or what is the object that it is pointing to.

So, the binding between the expression like pd → g and the functions B :: g() or D :: g(), this binding whether it binds here or it binds here does not depend on the pointer. It depends on the pointed object. It does not depend on pd it depends on what is the type of the pointed object. If the type of pointed object is of b type as it was here b type then, the g method of the base class is invoked. If it is of d type then the g method of the derived class gets invoked. And therefore, this is called the dynamic binding this is in contrast to

the static binding. And what makes the difference? The difference is made by this key word before the function. So, of the two functions this is called a virtual function where I have written the virtual in the front and this is called a non-virtual function.

And if I have a non-virtual function which is what we had earlier. I will have a static binding and if I have a virtual function then i will have a dynamic binding where the if I call that function through a pointer then it will not depend on the type of the pointer, but it will depend on the actual object type of the actual object that the pointer points to at the run time. So, that is a basic difference between static and dynamic binding of course, I am just trying to introduce the semantics to you. We have we are still a little way to go to understand to realize why we are doing this. How will this really help in terms of modeling an implementation, but this is this basic notion of virtual function and dynamic binding is what we want to understand from here.

And in the last section where we use the reference we can see exactly the same behavior. These two are the here the reference are b refers to the b object; rd refers to the d object. For function f it again if I do it through reference I have a static binding, but certainly if I invoke the g function g method for rb and rd because rd is referring to a d object and because rb is referring to a b object. According to dynamic binding the in this case, I get a get the g function of the derived class invoked. In this case I get the g function of the base class invoked. So, if I access the functions as object they will always be static. They will always be based on what that object type is. But if I invoke the methods through functions or through reference then I may have static or dynamic binding depending on whether the member function I am invoking is a non-virtual one where the static binding will happen or when the member function is a virtual one where dynamic binding will happen. So, this was just to introduce the basic notion of binding will build up further on this.

(Refer Slide Time: 29:15)

The slide is titled "Module Summary" and is part of "Module 26" by Partha Pratim Das. The slide content includes a bulleted list of key points:

- Introduced casting and discussed the notions of upcast and downcast
- Introduced Static and Dynamic Binding

The slide also features a small video thumbnail of the instructor, Partha Pratim Das, speaking.

So, to sum up we have introduced the concept of casting and discussed the basic notion of upcasting and downcasting and seen that up cast is safe and down cast is risky. We have done this because in the next module, we will need to use this notion of casting in the context of binding. After that we have introduced the basic semantics of static casting and dynamic casting or the basic definition of a virtual function which is a new kind of member function that we are introducing further classes. We will continue discussions on the dynamic binding in the next module.

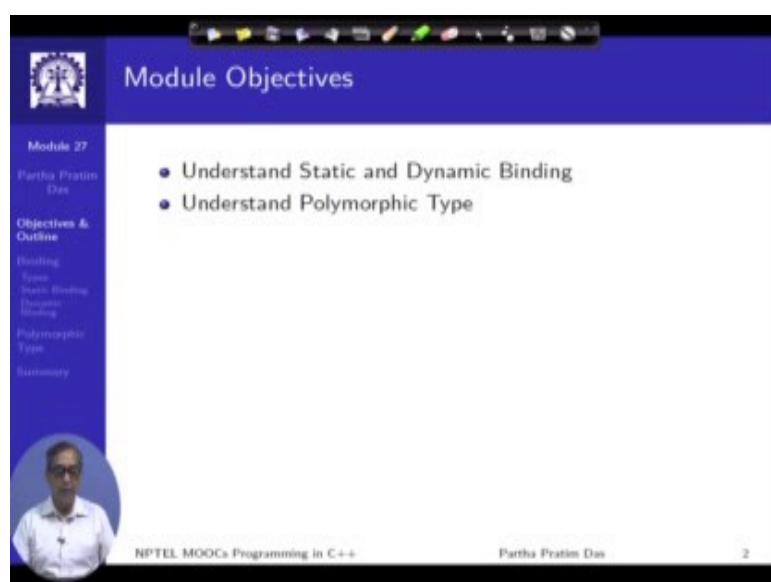
Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 42
Dynamic Binding (Polymorphism): Part 2

Welcome to Module 27 of Programming in C++. We have been discussing about static and dynamic binding. In fact, we talked at length in the last module about various casting options, particularly when casting is done on a class hierarchy; and we observed that if we upcast from a specialized class to a generalized class then that is a valid operation.

Because by interpreting specialized class object as a generalized class object we are only using part of the information that is available. But downcast when you try to cast a generalized class object as a specialized class object, we are trying to interpret information that does not exist for the specialized class, so that will be an error. And we had discussed about this casting issue, because particularly when we discuss dynamic binding over class hierarchies then regularly we need to perform a different kinds of upcast.

(Refer Slide Time: 01:37)



The screenshot shows a presentation slide titled "Module Objectives" for "Module 27". The sidebar on the left contains the following navigation links:

- Module 27
- Partha Pratim Das
- Objectives & Outline
 - Binding
 - Type
 - Basic Binding
 - Dynamic Binding
 - Polymorphic Type
 - Summary

A video player in the bottom-left corner displays a thumbnail of a man, identified as Prof. Partha Pratim Das. The main content area of the slide lists the "Module Objectives" as follows:

- Understand Static and Dynamic Binding
- Understand Polymorphic Type

So, we have taken a look at an example. What we will do in this module we will continue to discuss about or detailed understanding of static and dynamic binding; and with that, we will introduce what is known as a polymorphic type in C++.

(Refer Slide Time: 01:52)

Module Outline

- Binding
 - Types
 - Static Binding
 - Dynamic Binding
- Polymorphic Type

Module 27
Partha Pratim Das
Objectives & Outline
Binding
Polymorphic Type
Summary

NPTEL MOOCs Programming in C++
Partha Pratim Das

This will be the outline of the module, and will be visible on the left hand side of your screen always.

(Refer Slide Time: 02:02)

The slide is titled "Type of an Object". It features a sidebar with navigation links: Module 27, Partha Pratim Das, Objectives & Outline, Pointing Types, Static Binding, Dynamic Binding, Polymorphic Types, and Summary. A video thumbnail of the speaker is also present. The main content area contains a bulleted list of points and a code snippet. To the right, there is a hand-drawn diagram illustrating pointer types.

- The static type of the object is the type declared for the object while writing the code
- Compiler sees static type
- The dynamic type of the object is determined by the type of the object to which it currently refers
- Compiler does not see dynamic type

```
class A {};
class B : public A {};

int main() {
    A *p;
    p = new B; // Static type of p = A
                // Dynamic type of p = B
}
```

A hand-drawn diagram shows a rectangle labeled 'A *p;' with an arrow pointing to another rectangle labeled 'B'. Above this, the text 'A == p;' and 'p == new B;' is written.

Let us understand what we mean by the static and dynamic type of objects. We have so far understood that in object if I have a class A and I define an object of instantiate an object of class A, then the type of this the type of a is the class A. We have understood that and that is what is uniform all through. Things start becoming different when we have a pointer variable or a reference variable, which is of a certain type, certain class which is a part of a hierarchy. So here we are showing such an instance, we have a class A, and we have another class B, the hierarchy diagram is like this, which is a specialization of A, B ISA A.

And in this context, when we have a pointer of type A, so it means that it is this pointer p can store an address where I expect to get an A type of object which is fine. So if I just do say if I say that I have A^*p , and I have created a new object a, and kept the address in p so that will be in the memory somewhere this object a has been created dynamically, instance of A has been created dynamically instance of A has been created. B is the pointer, which is pointing to this. In this case, the type of p as we know as a compiler we will get to see is that of a pointer to A. And the object that it is actually pointing to is also of type A, which is the normal scenario.

(Refer Slide Time: 04:18)

The slide has a blue header with the title 'Type of an Object'. On the left, there's a sidebar with a logo, the text 'Module 27', 'Partha Pratim Das', 'Objectives & Outline', 'Pointing to Objects', 'Types', 'Static Binding', 'Dynamic Binding', 'Polymorphic Types', and 'Summary'. Below the sidebar is a video player showing a man in a white shirt. The main content area contains bullet points and code. The bullet points are:

- The static type of the object is the type declared for the object while writing the code
- Compiler sees static type
- The dynamic type of the object is determined by the type of the object to which it currently refers
- Compiler does not see dynamic type

The code shown is:

```
class A {};
class B : public A {};

int main() {
    A *p;
    p = new B; // Static type of p = A
                // Dynamic type of p = B
}
```

At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

But, let us think that instead of creating A object, I have created a B object. So I have created here, dynamically I have created a B object which means that it has a base part which is of A type, and this is my whole dynamically created object. And I have a pointer p, which is of type A, so the pointer p points to this. This we have seen this is a scenario of upcast that we have seen is quite possible; and therefore, actually if we look at the object through p then we will be able to see only the base class part of it we have understood why this is so. Now, in this context, we can see that the actual object that is created has a type which is of B; this has a type which is of B. And the pointer has a type which is of A.

So we will distinguish these two now. We will say that the type of p statically the type of p is A, why are we say statically, because this is what the compiler has got to see. The compiler has got to see that p has been defined as of A type pointer. But what has happened in the run time, what has happened in the actual execution; in the actual execution act, really a B type object has been created, and it is pointer put to p. So, the dynamic type of what p points to is actually, it is pointing to a B type object though it is actually A type pointer. So this is the notion of the dynamic type.

As I mentioned that if I had just done `A*p` and we had done a instantiated `A` object there then the static and dynamic type both are `A`, and they will be same, but it is possible that the static and dynamic type are different. So, it is very important to keep track of what is the static type that is what the compiler see and what is a dynamic type which happens at the run time with the compiler cannot see. Now, certainly you would be curious to know as to why we are creating these two static and dynamic type concepts, and how they will be used, so that will unfold after maybe another module also when we actually show applications of how the dynamic type is really very critical to do object oriented programming.

(Refer Slide Time: 07:01)

Module 27
Partha Pratim Das
Objectives & Outline
Hosting Types
Static Binding
Dynamic Binding
Polymorphic Type
Summary

Static and Dynamic Binding

- **Static binding (early binding):** When a function invocation binds to the function definition based on the static type of objects
 - This is done at compile-time
 - Normal function calls, overloaded function calls, and overloaded operators are examples of static binding
- **Dynamic binding (late binding):** When a function invocation binds to the function definition based on the dynamic type of objects
 - This is done at run-time
 - Function pointers, Virtual functions are examples of late binding

Handwritten notes in red ink on the right side of the slide:

- void void (int)
- void void (double)
- void void (string)
- void void (int)
- void void (double)
- void void (string)
- void void (int)
- void void (double)
- void void (string)
- void void (int)
- void void (double)
- void void (string)

Partha Pratim Das

So, with this type of object, we can define static and dynamic binding. So, formally, I just shown an example, but now I am giving you the formal definition. We say that when a function invocation binds to the function definition that is done based on the static type we say that a static binding has happened. So, what we are saying we are saying I have a function say `void f()` and I am calling `f()`, so while I am calling `f()`, I am binding to this `f()`.

Let say different scenario, I have `void f (int)`, I have `void f` that is I have overloads of this and then I am calling `f (3)` or I am calling `f()`, if I call `f (3)`, I bind to this; if I call `f()`, I

bind to this. So, binding is the process by which from an invocation, I can say what is the actual function that will be invoked from this invocation that is the process of binding that is what is known as binding.

And if I can resolve that at the compile time if I can decide on that at the compile time then I say that I have static binding, and alternately this is also called early binding, because early in the sense that the compilation certainly has to precede the execution of a program. So here at the compilation time at the program writing time itself I know what the binding will be, and therefore, this is also called early binding. So normal function calls as I was showing; overloaded function calls, overloaded operators and so on. The various different kinds of function invocations that we have seen are typically examples of static binding.

In contrast, the dynamic binding talks about when a function invocation binds to the function definition based on the dynamic type of the object that is if we have class B as a specialization of A, and let say both of them have a function f(), and I am invoking a function based on a pointer. I need to decide whether it should bind here and whether it or it should bind here. If this decision depends on the dynamic type of p that is not the type of p itself, p could simply be a type of A - the base type, a pointer to the base type.

But if the binding depends not only on the type of p, but on the actual object that the p is pointing to. If it is pointing to A type object, then it should bind to the f member of class A, but if p is pointing to a B type object then it should bind to the f member of the b object, this is what is known as dynamic binding. Certainly whether p points to an A type object or it points to a B type object, $p \rightarrow f()$ this expression does not change.

So, the compiler has got to see only $p \rightarrow f()$. So compiler, at the compilation time at the static time could not have decided as to whether $A :: f()$ will be invoked or $B :: f()$ will be invoked this will be decided later at the time of execution that is why this is called late binding. So this is dynamic binding, because it depends on the dynamics or the execution run time of the program. It is late binding because it is happening later than the compilation time. This is as it is the run time.

So, though it was not very formally discussed and organized, C also has this kind of a concept in terms of function pointers. So if I say that I have a typedef and I say void (* p f), then pf becomes f type of a function pointer. So I can say pf then myf then myf is a function pointer which can point to any of the functions which takes a void and returns a void. So, in this context, if I write void g(), if I write void h(), then if I then invoke myf() as a function then this expression may invoke g() or may it may invoke h(), depending on if I have assigned g() to myf then this will invoke g(). If I have assigned h() to myf then this will invoke h(). So this is the basic concept of the function pointer that you know.

And so this also is a situation of dynamic binding, because compiler from knowing this expression cannot know whether myf actually has been set to point to the g function or to point to the h function. So, function pointers also are the fundamental which offer the dynamic binding in C itself and certainly in C++. And then we will see that the virtual functions are the right example of late binding, dynamic binding that we have in C++.

(Refer Slide Time: 13:05)

The slide is titled "Static Binding" and is part of "Module 27". It features two code snippets side-by-side:

Inherited Method:

```
#include<iostream>
using namespace std;
class B { public:
    void f() {}
};
class D : public B { public:
    void g() {} // new function
};
int main() {
    B b;
    D d;
    b.f(); // B::f()
    d.f(); // B::f() ----- Inherited
    d.g(); // D::g() ----- Added
}
```

Overridden Method:

```
#include<iostream>
using namespace std;
class B { public:
    void f() {}
};
class D : public B { public:
    void f() {} // overrides B::f()
};
int main() {
    B b;
    D d;
    b.f(); // B::f()
    d.f(); // D::f() ----- Overridden
    // masks the base class function
}
```

Annotations on the slide include a red circle around the word "Inherited" in the first code snippet and a red arrow pointing from the "d.f()" line in the second code snippet to the note "masks the base class function".

At the bottom, there is a summary list:

- Object d of derived class inherits the base class function f() and has its own function g()
- Function calls are resolved at compile time based on static type
- If a member function of a base class is redefined in a derived class with the same signature then it masks the base class method
- The derived class method f() is linked to the object d. As f() is redefined in the derived class, the base class version cannot be called with the object of a derived class

Other visible text on the slide includes "Partha Pratim Das" and "NPTEL MOOCs Programming in C++".

So let us go over into different cases. So, first little bit more on the static binding. I have a class B, as a base - class d is the derived specialized class. So, if I have a member f here and a member g here, and then I construct two objects. If I do b.f() certainly it will call this function because it knows statically. If I call d.f(), it will also call this function,

this will also call this function, why because d has inherited, and we know that being a specialization it will inherit, so d has inherited f() so it will call B :: f() because it has been able to inherit that; though it is not explicitly written in the scope of d. And certainly if I call the new function g, d .g() then it will call the d .g(), which is so this is the basic notion of inherited functions that we had seen.

Now, if I overwrite, I am doing the same thing, but instead of introducing a new function I have introduced the signature of the same function in B. If I have done that then this will kind of, now if I do b.f() certainly, but now if I do d.f(), it will invoke the new D::f() or the overwritten function. So, moment I overwrite I am actually masking the base class function that is for d, the base class function B :: f() is no more available so that is a basic structure of the overriding operations. So, we will have to be careful in dealing with that. We cannot use both the base class as well as the derived class functions.

(Refer Slide Time: 14:58)

```

Module 27
Partha Pratim Das
Objectives & Outline
Reading Log
Static Binding
Dynamic Binding
Polymorphic Type Summary

Member Functions – Overrides and Overloads: RECAP (Module 22)

Inheritance
class B { // Base Class
public:
    void f(int i);
    void g(int i);
};

class D: public B { // Derived Class
public:
    // Inherits B::f(int)
    // Inherits B::g(int)
};

B bi;
D di;

bi.f(1); // Calls B::f(int)
bi.g(2); // Calls B::g(int);

di.f(3); // Calls B::f(int)
di.g(4); // Calls B::g(int)

Override & Overload
class B { // Base Class
public:
    void f(int);
    void g(int i);
};

class D: public B { // Derived Class
public:
    // Inherits B::f(int)
    void f(int); // Overrides B::f(int)
    void f(string); // Overloads B::f(int)
    // Inherits B::g(int)
    void h(int i); // Adds B::h(int)
};

B bi;
D di;

bi.f(1); // Calls B::f(int)
bi.g(2); // Calls B::g(int);

di.f(3); // Calls D::f(int)
di.g(4); // Calls B::g(int)

di.f("red"); // Calls D::f(string)
di.h(5); // Calls D::h(int)

● D::f(int) overrides B::f(int)
● D::f(string) overloads B::f(int)

```

Now in this context, I would point you to an example that we had taken here earlier which had shown the same thing. And you have a base class is the function f, which is overridden here, and is also overloaded here. So, the derived class has two functions f and f which takes an integer and f which takes a string, so when we make the derived class object and we invoke f with 3, it invokes the f(int) function, if we invoke with “red “

then it invokes the overloaded f function. So, you can override and overload at the same time, so this is what we have seen. So, I am just reminding of this because all this will now get mixed.

(Refer Slide Time: 15:53)

using Construct – Avoid Method Hiding

```
#include<iostream>
using namespace std;
class A { public:
    void f() { }
};

class B : public A {
    // To overload, rather than hide the base class function f()
    // is introduced into the scope of B with a using declaration
    using A::f();
    void f(int) { }
};

int main() {
    B b; // Function calls resolved at compile time
    // b.f(); // B::f(int)
    // b.f(); // A::f()
}
```

- Object b of derived class linked to with inherited base class function f() and the overloaded version defined by the derived class f(int), based on the input parameters – function calls resolved at compile time

NPTEL MOOCs Programming in C++ Partha Pratim Das

Now, suppose you have a situation, where you have a function in the base class, member function in the base class which you want to overload in the derived class. You have a function in the base class which you want to overload in the derived class. So you have written the overload.

Now the question is as you overload, you will also hide the inherited function that you have inherited from the base class. So, if you just do this, and then if you try to write b.f(), you will get an error, why will you get an error because now the compiler knows that you have just overloaded you have inherited, and overloaded that so the compiler knows that now you have f function in class b which is which will take an int and which will not work with no parameter. So, if you still want that you would like to inherit this, but unlike the earlier example would not like to override this, then you can make use of what is known as a using definition or using construct, so what you say using A :: f().

So what does it tell you, it tells you that you are inheriting this base class member function, and you have overloaded that, but as you inherit you do not want to override that; so with that, if you do `f b.f(3)` then it calls the B class function which is the overloaded 1, but if you just call `b.f()` without any parameter then it calls the inherited function. So we have seen that couple of things if we override we hide the function; if we overload, we also hide the inherited function; if we override the inherited function then we have a new function of the same signature in the derived class. But if we want that we would overload in the derived class, and also use the inherited function from the base class, then we can make use of this using construct.

So with that, all possible designed combinations can be done. And these are all decisions that are made in the static type so this is the different situations of static binding.

(Refer Slide Time: 18:29)

Dynamic Binding

Module 27

Partha Pratim Das

Objectives & Outline

- Binding
- Static Binding
- Dynamic Binding
- Polymorphic Type Summary

Non-Virtual Method

```
#include<iostream>
using namespace std;
class B { public:
    void f() {} };
class D : public B { public:
    void f() {} };
int main() {
    B b;
    D d;
    B *p;
    p = &b; p->f(); // B::f()
    p = &d; p->f(); // B::f()
}
```

- `p->f()` always binds to `B::f()`
- Binding is decided by the type of pointer
- Static Binding

Virtual Method

```
#include<iostream>
using namespace std;
class B { public:
    virtual void f() {} };
class D : public B { public:
    virtual void f() {} };
int main() {
    B b;
    D d;
    B *p;
    p = &b; p->f(); // B::f()
    p = &d; p->f(); // D::f()
}
```

- `p->f()` binds to `B::f()` for a B object, and to `D::f()` for a D object
- Binding is decided by the type of object
- Dynamic Binding

NPTEL MOOCs Programming in C++

Partha Pratim Das

Now, let us talk about dynamic binding. So, I again look into the static case; I have one member function here, which is specialized, which is overridden in the derived class function. So, if I have two objects, one of the base class and one of the derived classes. And if we try to put their addresses in a pointer of type base class, and invoke the function `f` as in here, then in both cases it will actually call the base class function. While it calls the base class functions because I am statically binding, because I know that `p` is a

pointer of type base class, which has this function. So if I do $p \rightarrow f()$, it will call this; if I do $p \rightarrow f()$, when p is actually pointing to a derived class object, it will still call the base class member, so that is the basic scenario.

We can change that by introducing the `virtual` keyword. If we say that this member function is `virtual`, and then if we override it in the, again we have the base class thus everything else is in this example except the fact that the function is now said to be a `virtual` one. In the same scenario, I have the same base class type pointer; I have the same two objects; and the two scenarios, where this points to the base class object in this it points to the derived class object. I am again trying to do $p \rightarrow f()$ look at statically these are the same expression, but if we do it with the base class object, it invokes the base class function; if I do it with a derived class object, it invokes a derived class function.

So, here this was both were going to the base class now the second one which actually has a dynamic type of being a `d` type of object, for that object the same pointer expression pointer invocation expression will take me to the derived class function. So, that is what dynamic binding is. So, you can see that the expression has not changed between these two, it is $p \rightarrow f()$, but depending on whether you are pointing to a derived class object or to a base class object, you are automatically being bound to either the derived class function or the base class function. So, this is the basic concept of `virtual` function or dynamic binding `virtual` methods that we have in C++, which will have several use in.

(Refer Slide Time: 21:18)

Module 27

Partha Pratim Das

Objectives & Outcome

Hosting Types

Static Binding

Dynamic Binding

Polyomorphic Type

Summary

Partha Pratim Das

NPTEL MOOCs Programming in C++

10

```
#include <iostream>
using namespace std;

class B {
public:
    void f() { cout << "B:f()" << endl; }
    virtual void g() { cout << "B:g()" << endl; }
};

class D: public B {
public:
    void f() { cout << "D:f()" << endl; }
    virtual void g() { cout << "D:g()" << endl; }
};

int main() {
    B b;
    B d;
    B *pb = &b; // UPCAST
    B *pd = &d; // UPCAST
    b.f();
    b.g();
    d.f();
    d.g();
    pb->f(); // B:f() -- Static Binding
    pb->g(); // B:g() -- Dynamic Binding
    pd->f(); // B:f() -- Static Binding
    pd->g(); // D:g() -- Dynamic Binding
    return 0;
}
```

So, this is just the example that we had seen in the last module. So, you could just go through this again for your understanding. So, we have one method, which is non-virtual; and we have other method which is virtual. And therefore, if we invoke all of these with the object then the respective member functions are invoked. So when we invoke from B the member functions of B are invoked; when I invoke from D member functions of D are invoked.

But when we use a base type of pointer to keep the address of either b or the address of d, of course, through an UPCAST, and then start doing that the same set of invocations through the pointer. Then for the base class object, I always invoke the base class member functions. But for the derived class object, I invoke the base class member function for a non-virtual method which is static binding which will take me here; but for a derived class object, I actually invoke the derived class member function because g is a virtual function g has a dynamic binding in contrast to f which has a non-dynamic or static binding.

And just to get clarify matters, the similar behavior would be shown if you use the instead of pointer, if you use reference as well. So these are to reference of the B class type which has alias to a b object and there is also a reference to a B class type, but it has

a alias to a d object through UPCAST. So, when I invoke g ()through this reference for the virtual function g, because actually I am maintaining the reference to a d object my invocation will go to the inherit will go the D class function the virtual function. So, this is the basic mechanism.

(Refer Slide Time: 23:32)

The slide has a blue header with the title "Polymorphic Type: Virtual Functions". On the left, there's a sidebar with "Module 27" and "Partha Pratim Das" at the top, followed by "Objectives & Outline" with several items listed, and a photo of a man in a white shirt. The main content area contains the following bullet points:

- Dynamic binding is possible only for pointer and reference data types and for member functions that are declared as virtual in the base class.
- These are called **Virtual Functions**
- If a member function is declared as virtual, it can be overridden in the derived class
- If a member function is not virtual and it is re-defined in the derived class then the latter definition hides the former one
- Any class containing a virtual member function – by definition or by inheritance – is called a **Polymorphic Type**
- A hierarchy may be polymorphic or non-polymorphic
- A non-polymorphic hierarchy has little value

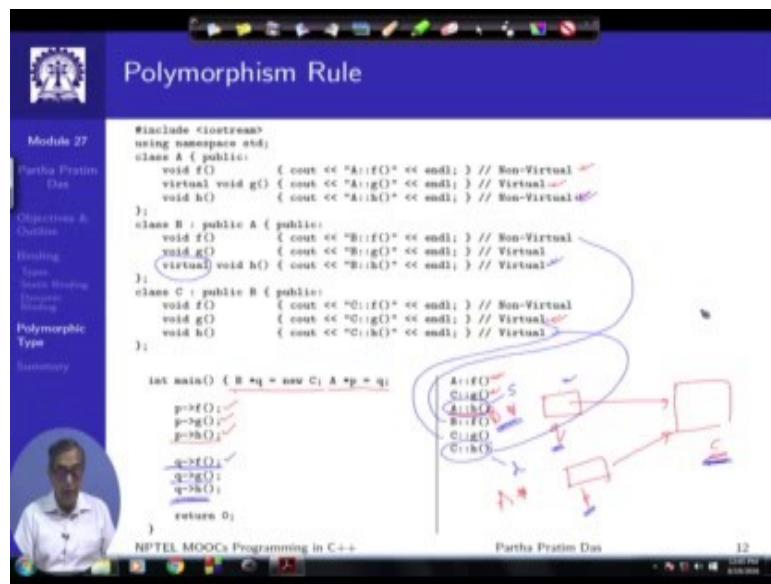
So, from based on this, we define the polymorphic type with the virtual function. We can see that the dynamic binding is possible only for pointer and reference data type. So, we have shown that if we directly invoke a member functions from the object then there will be bound statically always. Such functions which are called written as virtual in the front are known as virtual function, and if a virtual member function is declared virtual it can be overridden in the derived class as we have seen. If a member function is not virtual, and it is redefined in the derived class as we have say then the latter definition will hide or suppress mask the former definition.

So any class that contains virtual member function either by definition or by inheritance, you may have defined a virtual member function yourself or you may have inherited it from your parent, but if you have at least one virtual member function, then that class is said to be a polymorphic type. Polymorphic, because it can take different forms at based on the run time object that the pointer of the reference is referring to. And certainly a

hierarchy as a whole could be polymorphic or non-polymorphic depending on; if a hierarchy is non polymorphic then certainly it does not have any polymorphic function or any virtual function in terms of the classes that it involves.

But, if the base class of a hierarchy has a virtual function or some class has a virtual function then the whole hierarchy that hangs from that class will become polymorphic. And as we will see that non-polymorphic hierarchies are really I mean they can be created, but they just have structural value, but they will have a little computational value, because you will not be using the major advantage of having the polymorphic type.

(Refer Slide Time: 25:27)



Now, I would just highlight a little bit of rule on polymorphism. I am taking another example, where A is the base class; B is a specialization of that C in turn is a specialization of B. So, this is a simple multilevel inheritance. I have three functions f, g and h. So, in class A, this is just simply defined, so this is non-virtual. This is defined as virtual and this is another h is also defined as non-virtual. So, what I can say that this now has A has at least one polymorphic function or at least one virtual function, so this whole hierarchy is a polymorphic hierarchy as a first thing we observe.

Then going to class B all these functions are overridden, so B has overridden this which continues to be non-virtual. `g()` is again overridden in B which continues to be virtual this is what needs to be noted that once I make a function virtual in a class, then any class which derives it must get it as a virtual function. And for that it is not mandatory to write the `virtual` keyword here.

I may write that as I did in the last example, I may not write that. But even if I do not write the `virtual` keyword in front of this inherited function, which I am overriding, while that inherited function was virtual in the base class that it will continue to be virtual in the derived class as well. So, once `g` is polymorphic here, `g` is virtual here, irrespective of whether I simply inherit it or I inherit and override it. And I writing the `virtual` keyword in front of it may be a good practice, because anybody else can quickly understand that, but it is not mandatory.

Now, what I do I do something more interesting, I have also inherited `h()`, which was non-virtual and coming to B, I make it virtual that is now I have written it virtual. So, what will happen in C, if as C overrides them. This continues to be non-virtual, this was non-virtual here and both of these which were virtual in B will now become also virtual in C. So in view of this, if I if I try to see if I have in key in `q`, I create a C type of object, and I am using two pointers, one is a `q` pointer to point, and one is a `p` pointer to point.

The difference being `p` is of type ‘A’ type pointer, and `q` is a ‘B’ type pointer. So, what happens if I do `p → f()` then certainly it is pointing to a C type object, `p` has the static type of `p` is `a`, `f` is a non-virtual function in class A, so `A :: f()` gets called that is a simple static binding case.

But if I call `p → g()` then `g` is a virtual function in A, so the dynamic type of `p`, which is type C will be used, and therefore, this function which is the virtual function in C the overridden virtual function is C which will get called. And if I invoke `h()`, then naturally again it is like `f()`, it is non-virtual in A, so if I since I am calling from `p`, it is decided by the type of `p` and `A :: h()` the function in a will be invoked.

Now, let us consider q, let us consider the other pointer, I look $q \rightarrow f()$. What is f, f is non-virtual so this will be statically bound. So what it is non-virtual in B, because q is a B type pointer. So, now, I have to see what is the definition of the functions in B, because q is a B type of pointer. Now in q in B, f() is a non-virtual, so if I do $q \rightarrow f()$, it will call the f functions in B, because it is statically resolved. g is a virtual function in B, so it will do a polymorphic dispatch, it will be decided based on the type of object that q is pointing to which is C type object, therefore, this will invoke the g function in the C class because it is decided by the type of the object.

The interesting thing happens with when I do $q \rightarrow h()$. When I do $q \rightarrow h()$, q is of type B, so I will look up the h function in type B. In type B, h function is a virtual function; now since it is a virtual function, its invocation binding will be decided by the dynamic type that is the type of C. So, this $q \rightarrow h()$ now calls the h function in C. So, you can see that earlier, when I invoked for the same object, when I did the invocation from p, it invoked this function A : $:h()$. When I invoked it from using the pointer to B, it invokes the C : $:h()$. So, here this was static and here this has become dynamic. And the change has happened because in the derived class B, I have changed while overriding, I have changed the binding property of the function h.

So please study this example in further detail and try to understand the reason that this is a basic polymorphic rule that once a virtual it will continue to be virtual in all specializations. But any non-virtual function, at any stage could be made virtual, and from that point onwards, downwards in the hierarchy it will continue to be a virtual function. And the compiler will always take the static type of the pointer and go to that class see what is the function whether the function is virtual or non virtual. If it is non-virtual, it will use static binding; if it is virtual, it will create code for dynamic binding, so that the binding will be decided based on the actual type being used at the run time.

(Refer Slide Time: 32:30)

The slide is titled "Module Summary" and is part of "Module 27" by Partha Pratim Das. The slide content includes a list of key points:

- Static and Dynamic Binding are discussed in depth
- Polymorphic type introduced

The sidebar on the left lists navigation options: Objectives & Outcome, Pointing Types, Static Binding, Dynamic Binding, Polymorphic Type, and Summary.

At the bottom, the footer reads: NPTEL MOOCs Programming in C++ Partha Pratim Das 13

To summarize, we have taken a deeper look into static and dynamic binding, and tried to understand the polymorphic type. And in the next module, we will continue our discussions on various specific issues that arise with the polymorphic types.

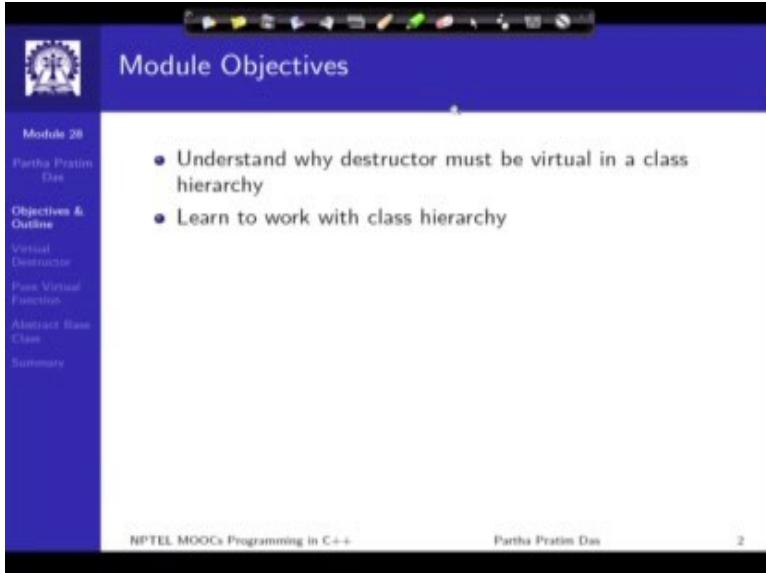
Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 43
Dynamic Binding (Polymorphism): Part III

Welcome to Module 28 of Programming in C++. We have been discussing about Static and Dynamic Binding. In this context in module 26, we discussed about casting particularly on a hierarchy up cast and down cast issues. And then in the last module, we have formally introduced a Notion of Binding. While I will invoking a function use in a pointer or a reference, how does the compiler resolve the actual function that will be invoked, whether it does it based on static strategy which will be done for non-virtual functions or it invokes involves a dynamic strategy for virtual functions. And we have seen what are the basic rules of static and dynamic binding that is engaged in C++.

And we have seen that whenever a particular class has a virtual function whether that function is inherited or that function is introduced in that class it becomes a polymorphic type. And based on that, we will be typically discussing about polymorphic hierarchies and thoroughly try to illustrate to you as to how they become very useful modelling and programming tool in terms of the object oriented paradigm.

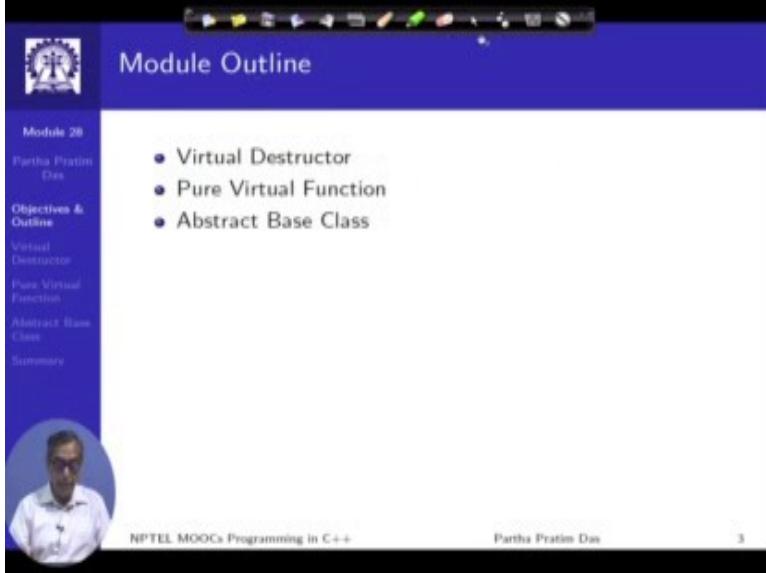
(Refer Slide Time: 01:41)



The slide has a blue header bar with the title "Module Objectives". On the left, there is a sidebar with the "NPTEL MOOCs Programming in C++" logo, the text "Module 2B Partha Pratim Das", and a "Objectives & Outline" section containing links to "Virtual Destructor", "Pure Virtual Function", "Abstract Base Class", and "Summary". The main content area contains two bullet points: "Understand why destructor must be virtual in a class hierarchy" and "Learn to work with class hierarchy". At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" with a page number "2".

In this particular module, we will continue on that polymorphic type. And we specifically try to understand why destructors must be virtual in a polymorphic hierarchy. And we will also try to start learning as to how to work with the polymorphic hierarchy.

(Refer Slide Time: 02:03)



The slide has a blue header bar with the title "Module Outline". On the left, there is a sidebar with the "NPTEL MOOCs Programming in C++" logo, the text "Module 2B Partha Pratim Das", and a "Objectives & Outline" section containing links to "Virtual Destructor", "Pure Virtual Function", and "Abstract Base Class". Below the sidebar is a circular portrait of a man. The main content area contains three bullet points: "Virtual Destructor", "Pure Virtual Function", and "Abstract Base Class". At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" with a page number "3".

So, in a specific we have three topics to discuss which form our outline and we will be visible on the left hand side.

(Refer Slide Time: 02:12)

```

Module 2B
Partha Pratim Das
Objectives & Outline
Virtual Destructor
Pure Virtual Function
Abstract Base Class
Summary

Virtual Destructor
#include <iostream>
using namespace std;

class B {
public:
    int data;
    B(int d) { cout << "B(" << d << ")" << endl; }
    ~B() { cout << "~B()" << endl; }
    virtual void Print() { cout << data; }
};

class D : public B {
public:
    int *ptr;
    D(int d1, int d2) : B(d1), ptr(new int(d2)) { cout << "D(" << endl; }
    ~D() { cout << "~D()" << endl; delete ptr; }
    void Print() { B::Print(); cout << " " << *ptr; }
};

int main() {
    B *p = new B(2);
    B *q = new D(3, 5);

    p->Print(); cout << endl;
    q->Print(); cout << endl;

    delete p;
    delete q;

    return 0;
}

```

Output:
B()
B()
B()
2
3 5
~B()
~B()

Destructor of d (type D) not called!

So, first is a Virtual Destructor. So, let us start with an example. So, this is the example. There is a class B, the base class, which has some int type of data. There is a class D which specializes from B and it is a pointer to integer type of data. Look at the constructor, the constructor simply takes a value and assigns sets to the member. The destructor does nothing it. We have put messages in the construct and destructor, so that we can trace and understand what is going on. Similarly, if I have a the derived class constructor, it takes two numbers; first it uses to construct the base part, calls a base part constructor; and the second, it uses to initialize its own pointer data, it dynamically allocates an integer with the value of D 2 and sets a pointer to the ptr member. It also has a message to say that it has happened.

Coming to the destructor of the derived class, we have a message saying that the destructor is being used. And certainly since now the object is about to be destroyed; it had a dynamic allocation in the ptr pointer, so that allocation will have to be released. So, we do a delete here. In addition to be able to see what is inside the classes, we use a use print functions and we make the print function virtual, so that it can be invoked from the

pointer and depending on the object type, it will do an appropriate print of either the data or the data and the pointer. Since, data is private here the overridden print function in D we cannot actually access data. So, this function I cannot write cout; even though data is a member of the D class, I cannot write this because data is private here, and therefore it cannot be accessed.

So, what we do is we follow a simple trick. We in turn invoke the corresponding member function in the B class. So, I directly invoke B : :print(), which means that it will invoke this member function print of the B class with the this pointer of the derived class D. And since this invocation actually needs this pointer of type B, what will happen is we are going up the hierarchy, so an upcast will automatically happen. So, that is what is the meaning of this. So, this is about the code that you get to see you can go through further literature.

(Refer Slide Time: 05:02)

The screenshot shows a presentation slide with the title 'Virtual Destructor'. On the left, there is a sidebar with navigation links: 'Module 2B', 'Partha Pratim Das', 'Objectives & Outline', 'Virtual Destructor', 'Pure Virtual Function', 'Abstract Base Class', and 'Summary'. Below the sidebar is a video player showing a person speaking. The main content area contains C++ code and its output. The code defines classes B and D, where D overrides the print function from B. In the main function, two objects of type D are created and printed, followed by their deletion. A note at the bottom right states 'Destructor of d (type D) not called!'. To the right of the code, the output is shown as a tree diagram with nodes labeled 'B()', 'B()', 'B()', '2', '3', '5', and 'B()', 'B()'. The number '4' is also visible in the bottom right corner of the slide.

```
#include <iostream>
using namespace std;

class B {
    int data_1;
public:
    B(int d) : data_(d) { cout << "B(" << d << ")";
    }
    virtual void Print() { cout << data_1; }
};

class D : public B {
    int *ptr_1;
public:
    D(int d1, int d2) : B(d1), ptr_(new int(d2)) { cout << "D(" << d1 << ")";
    }
    ~D() { cout << "~D()" << endl; delete ptr_1; }
    void Print() { B::Print(); cout << " " << *ptr_1; }
};

int main() {
    B *p = new B(2);
    B *q = new B(3, 5);
    p->Print(); cout << endl;
    q->Print(); cout << endl;

    delete p;
    delete q;

    return 0;
}
```

Outputs:
 B()
 B()
 B()
 2
 3 5
 B()
 B()

Destructor of d (type D) not called!

Partha Pratim Das

4

And now then in the application we create a B class object, we create a D class object put them to two pointers, and we print. And certainly if we do that from the creation, this is the construction of the B class object that is pointed to by p. These two come from the construction of the D class object because first B part of the D class object is constructed then D part is, then you do print you see 2, 3, 5 as you see this member, so up to this

point there is nothing interesting. So, forget about this was all setting up. And then we have the delete for these pointers. We want to basically delete the object. So, if I delete this what will happen this should invoke my B class destructor which it does, so that that is pretty fine. Finally, I invoke the destructor of invoke delete on q which should invoke the destructor of D and see what I get. I do not get the destructor of D, I do not get this printed fine. So, why is it that?

Now if we apply our mind and think about what is going on in terms of the binding see this destructor, destructors are in a way are member functions which are defined here as non-virtual. So, what will happen when I want to call delete on q then it has to decide based on what is the type of q, is that is what the compiler we has to see. So, compiler wants to decide based on q and q is of type B. So, it has to go to class B and decide that what destructor should be called it knows there is a destructor and it finds that this destructor is non-virtual. So, it calls it done. The destructor of D will never get called, because actually what we needed to be able to destroy this object, we needed the call to land in the destructor of D which in turn would call the destructor of B and do things right that is how it happens. So, here the call landed at a wrong place. So, this is the basic problem of a destruction way for which we need to make the destructor virtual.

(Refer Slide Time: 07:40)

The screenshot shows a presentation slide with the title "Virtual Destructor". The slide content is as follows:

```
#include <iostream>
using namespace std;

class B {
    int data_1;
public:
    B(int d) : data_(d) { cout << "B(" << d << ")" << endl; }
    ~B() { cout << "~B()" << endl; }
    virtual void Print() { cout << data_1; }
};

class D : public B {
    int *ptr_1;
public:
    D(int d1, int d2) : B(d1), ptr_(new int(d2)) { cout << "D(" << d1 << ", " << d2 << ")" << endl; }
    ~D() { cout << "~D()" << endl; delete ptr_; }
    void Print() { B::Print(); cout << " " << *ptr_; }
};

int main() {
    B *p = new B(2);
    B *q = new B(3, 5);

    p->Print(); cout << endl;
    q->Print(); cout << endl;

    delete p;
    delete q;
    return 0;
}
```

Output:

```
B()
B()
B()
D()
3 5
~B()
~B()
```

Destructor of d (type D) not called!

The slide also includes a sidebar with the following navigation links:

- Module 28
- Partha Pratim Das
- Objectives & Outcome
- Virtual Destructor** (highlighted in red)
- Pass Virtual Function
- Abstract Base Class
- Summary

A circular red arrow highlights the code block, and another red arrow points from the output text to the last line of the code: "delete q;"

So, it is the fix is very simple. All that you do is write the word virtual between in front of the destructor of p; up to this point there is nothing different, there is nothing different in the expected behavior and the output. Now what happens in if you to delete q, q is of type B, so it goes to class B, the compiler sets it to class B. In class B, you find that the destructor is virtual which means now the dispatch has to happen not based on the type of the pointer, but the type of the object it is pointing to. And what is the type of the object, what is a dynamic type the dynamic type is D. So, when we do this, actually this gets involved this does not this invokes the destructor of D, which in turn will invoke the destructor of B. So, you see that the destructor of D is invoked this, this gets deleted and then at this point the destructor of B is invoked. So, you get to see the B has got invoked. By the rule of polymorphism hierarchy that we have seen earlier, since the destructor of B is virtual by inheritance the destructor of D is also virtual. I do not need to write that it is virtual.

(Refer Slide Time: 08:58)

The screenshot shows a video player interface with a blue header bar. The title 'Virtual Destructor' is displayed. On the left, a sidebar lists 'Module 2B', 'Partha Pratim Das', 'Objectives & Outcomes', 'Virtual Destructor', 'Pure Virtual Function', 'Abstract Base Class', and 'Summary'. A small video thumbnail of the speaker is visible. The main area contains the following C++ code:

```
#include <iostream>
using namespace std;

class B {
    int data_1;
public:
    B(int d) : data_(d) { cout << "B()" << endl; }
    virtual ~B() { cout << "~B()" << endl; } // Destructor made virtual
    virtual void Print() { cout << data_1; }
};

class D : public B {
    int *ptr_1;
public:
    D(int d1, int d2) : B(d1), ptr_(new int(d2)) { cout << "D()" << endl; }
    ~D() { cout << "~D()" << endl; delete ptr_; }
    void Print() { B::Print(); cout << *ptr_1; }
};

int main() {
    B *p = new B(2);
    B *q = new B(3, 5);

    p->Print(); cout << endl;
    q->Print(); cout << endl;

    delete p;
    delete q;
}

return 0;
}
```

Handwritten notes are overlaid on the slide, including a diagram of a pointer pointing to a box labeled 'B()', and the word 'Slice' written vertically.

Output:

```
B()
B()
B()
2
3 5
~B()
~B()
~B()
```

Destructor of d (type B) is called!

NPTEL MOOCs Programming in C++

But without that the major problem that I was getting into that if this virtual were not there then this was not getting called; and the consequence of that, the pointer that was created in the d object was not being released. So, I can say that when I have an object from a derived class, which has a base class part. And I am holding that from a base class pointer, if the destructor is not virtual then it is simply looking at the base class part of

the pointer. So, it only deletes this part of the object. It does not delete the remaining part of the object. So, we say that the destructor tries to slice the object; it tries to chop off that object at this point, and only does releases one part. So, this is something very, very dangerous because we will have certain you know chopped off sliced object remaining in the system, and the system consistencies will go here where. So, if you are on hierarchy then make sure that the destructor in the base class will be a virtual function.

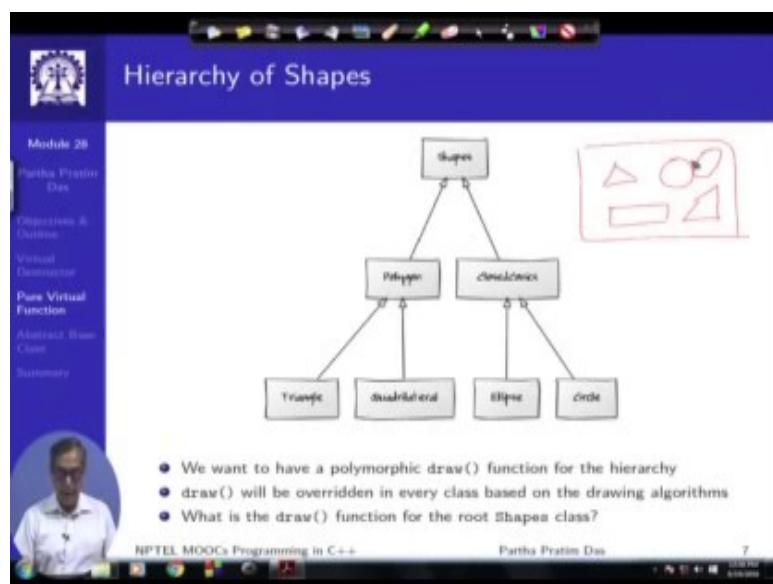
And now you can understand the moment you make the destructor of the base class virtual, because otherwise this whole mechanism will not work, the whole cleanup consistency will not work. But as soon as you make the destructor of the base class virtual, that means, that the base class becomes polymorphic irrespective of whether you have another polymorphic function like this or not; like here, we had another polymorphic function. But even if we did not have that, even if we did not have this print function, the moment I make this virtual class B becomes a polymorphic type, and therefore, since it is the root the all classes that are derived from that directly or indirectly all become polymorphic. So, the whole hierarchy becomes polymorphic. So, this is one of the reasons that I had mentioned earlier that if I have a hierarchy then it does not make sense to make it non-polymorphic, it is of not much of interest.

(Refer Slide Time: 11:05)

The screenshot shows a presentation slide with a blue header bar containing the title 'Virtual Destructor'. Below the header, there is a sidebar on the left with a circular profile picture of a man and a vertical list of navigation links: 'Module 28', 'Partha Pratim Das', 'Objectives & Outline', 'Virtual Destructor' (which is bolded), 'Post Virtual Functions', 'Abstract Base Class', and 'Summary'. The main content area contains two bullet points: '• If the destructor is not virtual in a polymorphic hierarchy, it leads to **Slicing**' and '• **Destructor must be declared virtual in the base class**'. At the bottom of the slide, there is footer text: 'NPTEL MOOCs Programming in C++', 'Partha Pratim Das', and a small number '6'.

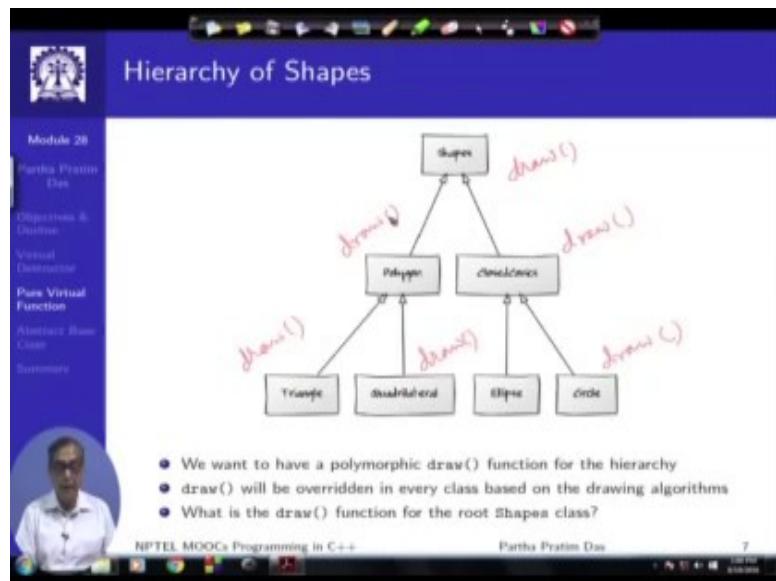
So, what we learn is if the destructor is not virtual in a polymorphic hierarchy it leads to slicing. So, destructor must be declared virtual in the base class always.

(Refer Slide Time: 11:16)



Let us look into some other interesting problem. Let us consider a hierarchy of shapes and our basic objective is we are trying to build a graphic system where these shapes can be there on the canvas. So, it is as if there is a canvas, and on that canvas, I want to draw objects of this shape different kinds of objects of this shape and so on, so that is a in deep objective. And there could be several other graphic things that we should be doing, but based on that we create a hierarchy. So, there is a shape, there are two kinds of shape that is basically polygonal shapes and closed curves like closed conics, there could be others also. In polygon, we have triangle, quadrilateral; there could be many more in closed conics, we have ellipse, circle and so on. So, this is just a simple.

(Refer Slide Time: 12:12)



And what we want to do is we want to have a polymorphic draw function here, because the way you draw a triangle say here drawing a triangle is basically drawing three segments of line. But drawing a circle is a pretty involved operation it uses some equation some algorithm drawing an ellipse is even more complicated, but in contrast drawing a quadrilateral would be. So, we want to have a hierarchy, where we have a draw function everywhere, so that we can simply we may not really need to bother about which particular object we have we will just hold a pointer to that object and call draw. And with that, we should be able to land by the use of dynamic binding or you should be able to land with the right draw function of the corresponding class, so that is what we want to do.

So, you want to have a polymorphic draw function for the hierarchy the draw will be overridden in every class based on the drawing algorithm. Now we get stuck because if I have to have this then certainly I need the draw function in the route, I need a draw function in the shape class. So, but the question is if I just a shape can you draw it, it is not possible to draw in arbitrary shape. In fact, which is words is for an arbitrary shape we cannot even represent it. So, we lead need some mathematical curves, some definitions to be able to represent shape, so that is the genesis of the problem that we are trying to address.

(Refer Slide Time: 13:39)

The screenshot shows a presentation slide titled "Pure Virtual Function". The slide content is as follows:

- For the polymorphic hierarchy of Shapes, we need draw() to be a virtual function
- draw() must be a member of Shapes class for polymorphic dispatch to work
- But we cannot define the body of draw() function for the root Shapes class as we do not have an algorithm to draw an arbitrary shape. In fact, we cannot even have a representation for shapes in general!
- Pure Virtual Function solves the problem
- A Pure Virtual Function has a signature but no body!

The slide also includes a sidebar with navigation links: Module 2B, Partha Pratim Das, Objectives & Outcome, Virtual Destructor, Pure Virtual Function, Abstract Base Class, and Summary. At the bottom, there is a photo of the speaker, Partha Pratim Das, and the text "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

So, for polymorphic hierarchy of shapes we need a draw to be a virtual function, draw() must be a member of shapes class, so that the polymorphic dispatch can work. So that I can have a pointed to the shape class type which can hold actual object instances of any of the different types of object that exist on the hierarchy of a triangle, of a rectangle, of anything. And we should be able to just from the pointed to shape, we should be able to just invoke draw and it should end up polymorphically dispatch to the particular draw function of the particular class of which the object I am pointing to. But certainly we cannot write the body of the draw function in the shape class, because we do not know the algorithm. So, for that a new notion is introduced which is called the pure virtual function. And a pure virtual function is characterized by it has a signature, but no body. We will see some exceptions to that, but the basic notion is it has the signature, so that I can call it, but it does not have a body because I cannot implement it; sounds weird, but let us see how does that fit in.

(Refer Slide Time: 14:49)

The screenshot shows a presentation slide titled "Abstract Base Class". The slide content is as follows:

- A class containing at least one Pure Virtual Function is called an **Abstract Base Class**
- Pure Virtual Functions may be inherited or defined in the class
- No instance can be created for an **Abstract Base Class**
- Naturally it does not have a constructor or the destructor
- An **Abstract Base Class**, however, may have other virtual (non-pure) and non-virtual member functions as well as data members
- Data members in an **Abstract Base Class** should be protected. Of course, private and public data are also allowed
- Member functions in an **Abstract Base Class** should be public. Of course, private and protected methods are also allowed
- A **Concrete Class** must override and implement all Pure Virtual Functions so that it can be instantiated

The slide also includes a sidebar with navigation links: Module 2B, Partha Pratim Das, Objectives & Outcome, Virtual Destructor, Pure Virtual Functions, Abstract Base Class, Summary. At the bottom, there is a video player showing a person speaking.

Now, if I have a pure virtual function in a class, at least one, then I call that to be an abstract base class. Now this virtual function - pure virtual function may be inherited or might be defined in the class, it does not matter which I get. But if a class has a pure virtual function, it is an abstract base class, what does that mean, what is abstract about this. So, the third point says it all. No instance can be created for an abstract base class; an abstract base has everything, but it cannot create an object instance. Why it cannot create an object instance. Conceptually, it is very clear because if I have a pure virtual function of which I just know the signature, I do not know the body, I do not know the algorithm, if I could create an object of the abstract base class then using that object I could invoke the pure virtual function which is a member of that abstract base class, but I do not have an algorithm for that so what do I execute. So, the way we restrict we say that the no instance can be created. Since, no instance can be created an abstract base class does not have a constructor or a destructor.

But it can have other virtual functions, it can have more pure virtual functions, it can have non-pure other virtual functions, it may have non-virtual functions, it may have data members and so on and so forth. Naturally, if it has data members then we would expect the data members to be protected, but it is possible that you have them as private and public I am saying it should be preferably protected because you do not expect an

instance of this class. So, if you do not expect an instance of this class, you do not expect that the class itself is doing some computation using the data members. So, the data members are there, so that the derive classes can use them. So, it is reasonable that they should be protected. Of course, you can meet them public, but that violates the basic encapsulation rule.

Similarly, the member functions of the class should normally be public, because certainly we will not have an instance, so if you do not have an instance then certainly it is a derive classes the hierarchy that we are going to use these functions. But, it is possible that you can have private or protected methods as well; and do different kind of tricks to hide the encapsulation. And since, we are talking about an abstract base class in contrast there are concrete class that must override an implement all pure virtual functions. Because now if you derive from, if you specialize from an abstract base class, naturally in the derived class, you do not have a default you will inherit the function, you will inherit the pure virtual function. But again in the derive class, you do not have an option of invoking that function, because that function by itself does not have a body. So, you will have to be able to create objects finally, you must have some derived classes which override the pure virtual function as non-pure virtual function, and implements them that is provides the body, such classes will be known as concrete classes. So, certainly for concrete classes, the instances can be created. This was a lot of set of rules.

(Refer Slide Time: 18:17)

The screenshot shows a video player window titled "Shape Hierarchy". The video content displays a C++ code snippet illustrating class inheritance and pure virtual functions. The code defines an abstract base class "Shapes" with a pure virtual function "draw()". It then derives concrete classes "Polygon" and "ClosedConics" from "Shapes". "Polygon" overrides "draw()", while "ClosedConics" inherits it and adds its own implementation. Subclasses "Triangle" and "Quadrilateral" inherit from "Polygon", and "Circle" and "Ellipse" inherit from "ClosedConics". All subclasses implement their own "draw()" methods. A main function creates four objects and calls their "draw()" methods. The video player interface includes a navigation bar with icons for back, forward, and search, and a progress bar indicating the slide number 10.

```
#include <iostream>
using namespace std;

class Shapes {
public:
    virtual void draw() = 0; // Pure Virtual Function
};

class Polygon : public Shapes { // Concrete Class
public:
    void draw() { cout << "Polygon: Draw by Triangulation" << endl; }
};

class ClosedConics : public Shapes { // Abstract Base Class
public:
    // draw() inherited - Pure Virtual
};

class Triangle : public Polygon { public:
    void draw() { cout << "Triangle: Draw by Lines" << endl; }
};

class Quadrilateral : public Polygon { public:
    void draw() { cout << "Quadrilateral: Draw by Lines" << endl; }
};

class Circle : public ClosedConics { public:
    void draw() { cout << "Circle: Draw by Bresenham Algorithm" << endl; }
};

class Ellipse : public ClosedConics { public:
    void draw() { cout << "Ellipse: Draw by ..." << endl; }
};

int main() {
    Shapes *arr[] = { new Triangle, new Quadrilateral, new Circle, new Ellipse };

    for (int i = 0; i < sizeof(arr) / sizeof(Shapes *); ++i) arr[i]->draw();
    // ...
    return 0;
}
```

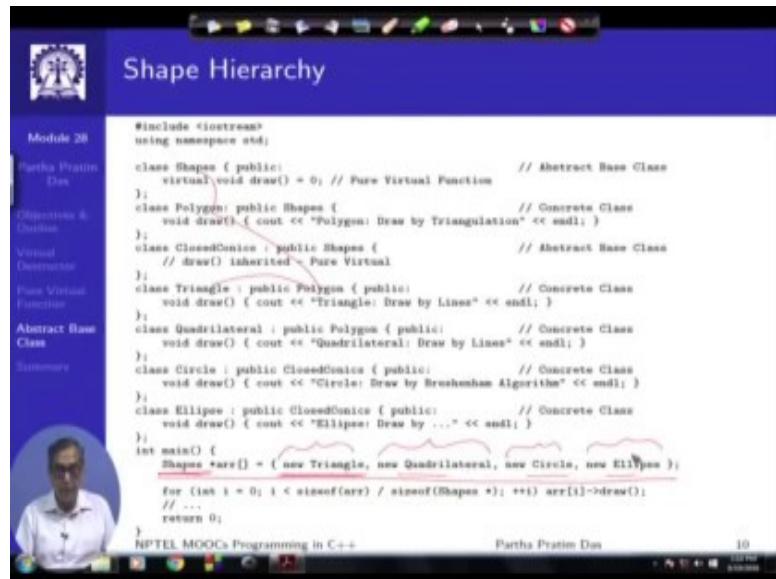
So, let us go into an example and try to understand. So, I have shown the hierarchy of shapes. At this point, you have feeling little lost in terms of the hierarchy, it will good to take a print out of that slide and keep it by the side, so that you can quickly refer to it. I did not have enough space to show it together, but it is conceptually very clear at the root class, base class in shapes. So, it has one virtual function draw, which does not take anything, does not return anything, but it is a virtual function. And we use this special notation of saying the '=' symbol then zero, which says that it is a pure virtual function. So, if you do not have this, then it is just a virtual function, but when you put this, this is called a pure virtual function which means that it is not expected now that there you will need an implementation for this function for the whole code to run. And since I have a pure virtual function, this shape becomes an abstract base class that is the basic.

Now, let us go down the shape had two specializations; one is polygon, and one is closed conics. Now in polygon, as I inherit we override the draw function, and polygon now has an implementation. Of course, this in a print message is just indicative of the implementation. So, possibly you implement an algorithm that the polygon is triangulated, and every triangle is drawn possible that way whatever. But, the point is we have now inherited and implemented the pure virtual function. So, this becomes an ordinary virtual function now. And therefore, the class becomes a concrete class that is it

will be possible to create instances of the polygon class. Look at the other class specialize from shapes, which is closed conics. In this, if it a closed conics, if you just say it a closed conic, then it could be a circle, it could be ellipse. So, I do not really know how to draw them through a generic algorithm. So, the closed conic also does not have a draw function. So, what it does, it does not override, it does not just have any signature of the draw function. So, what will happen, since this is a specialization of shapes from the shapes, it inherits the draw function, which is purely virtual in shapes, therefore, it is purely virtual in closed conics also, because no implementation has been provided. So, closed conic also continue to be the abstract base class.

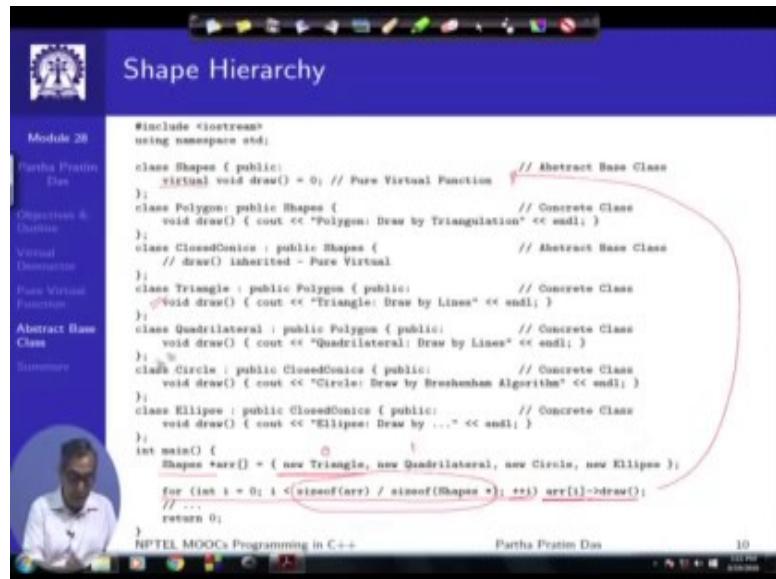
Then you have triangle specializing from polygon, quadrilateral specializing from polygon, as they specialize they have their own implementation. So, they are inheriting and overriding the function, so these are the specific draw functions for triangle quadrilateral base of classes. And then in terms of the other side, in terms of the closed conics, circle specializes from closed conics, ellipse specialize from closed conics, and they override the draw function with the circle specific drawing algorithm or ellipses specific drawing algorithm and so on. So, they become concrete class. So, now, as it turns out that the route is an abstract base class, the closed conics are the abstract base class, because both of them have one pure virtual function draw. All other five classes have become concrete, because each one of them have introduced an implementation have promise an implementation of the inherited draw function, which it has overridden.

(Refer Slide Time: 22:15)



So, with this now, if we look at the way we could create the canvas. So, the canvas say is now an array of a pointer to shape, so these are pointer to shape. And I have different kinds of shape, I have a triangle, I have a quadrilateral, I have a circle, I have an ellipse. So, I create all of these of course, in actual code, there will be lot of parameters and all that, but I am just demonstrating the whole process. So, I create pointers to variety of different objects; each one of them is eventually a specialization of shape. So, when I create a dynamically create a triangle, I get a pointer of a triangle type. And since triangle is a polygon, and a polygon is a shape, so I can upcast this pointer to the shapes pointer I can do that for each one of these pointers and they are all shape.

(Refer Slide Time: 23:14)



So, now, I have uniform array of pointers of shape through which I can actually invoke methods of all of these functions. Then I simply to draw I simply write a for loop, but I start from the zero, this code must be familiar with you this basically tries to find out how many elements are there in the array of shapes and you go over them. So, if we have the ith element arr[i] this of type shape pointer. So, on that I invoke draw function. So, when i will be 0, at it will actually have the value of new triangle. So, this will this call will take me the compiler will map it here first, because arr is of shape pointer type and then it finds that this is virtual, so that is to be dynamic dispatch. So, the dispatch will happen based on the actual object it is pointing to. For 0, this will be triangle. So, this particular function will get called. Then when becomes, so this for 0, for 1, this is a quadrilateral. So, the same dispatch will happen and this function will get called and so on.

(Refer Slide Time: 24:23)

The screenshot shows a presentation slide titled "Shape Hierarchy". On the left, there is a vertical sidebar with a logo at the top, followed by a list of navigation items: Module 2B, Partha Pratim Das, Objectives & Outcome, Visual Diagrams, Pure Virtual Functions, Abstract Base Class, Summary, and a photo of a man (Partha Pratim Das) in a blue circle. The main content area has a dark blue header with the title "Shape Hierarchy". Below the header, there is C++ code for a main() function that creates an array of shapes and calls their draw() methods. The output of this code is shown below the code. A bullet point on the right side states: "Instances for class Shapes and class ClosedConics cannot be created". At the bottom of the slide, there is footer text: "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

```
int main()
{
    Shapes *arr[] = { new Triangle, new Quadrilateral, new Circle, new Ellipse };
    for (int i = 0; i < sizeof(arr) / sizeof(Shapes *); ++i) arr[i]->draw();
    // ...
    return 0;
}

Output:
Triangle: Draw by Lines
Quadrilateral: Draw by Lines
Circle: Draw by Bresenham Algorithm
Ellipse: Draw by ...
```

- Instances for class Shapes and class ClosedConics cannot be created

So, if we try to quickly look at the output then for this code, the output will look like this. So, first was the new triangle, so in the output you see. The triangle draw has been called then the quadrilateral draw has been called, then the circle draw has been called, the ellipse draw has been called and so on. And all that we could do we needed to do is just call the function at the root, the polymorphic function at the root. And because of the ability to introduce pure virtual function and abstract base classes, we have been able to do this whole module together; otherwise, without that there is no way to generalize between the polygons and the closed conics into a same hierarchy we would have required actually closed conics itself is cannot be generalize in that way.

So, this is the basic advantage, the basic way you start using the polymorphic hierarchy to be able to write codes which are extremely compact in nature and we will see lot of other benefits like extensibility and so on. And but certainly for this case the instances of the shape and the closed conic classes cannot be created because they are abstract base class.

(Refer Slide Time: 25:35)

The slide has a blue header with the title 'Shape Hierarchy: A Pure Virtual Function may have a body!'. On the left, there's a vertical sidebar with navigation links: Module 2B, Partha Pratim Das, Objectives & Outcome, Visual Diagrams, Pure Virtual Functions, Abstract Base Class, Summary, and a photo of the speaker. The main content area contains C++ code demonstrating a shape hierarchy with pure virtual functions:

```
#include <iostream>
using namespace std;
class Shape { public:
    virtual void draw() = 0; // Pure Virtual Function
    { cout << "Shapes: Init Brush" << endl; }
};
class Polygon : public Shape { // Concrete Class
    void draw() { Shape::draw(); cout << "Polygon: Draw by Triangulation" << endl; }
};
class ClosedConvex : public Shape { // Abstract Base Class
    // draw() inherited - Pure Virtual
};
class Triangle : public Polygon { public: // Concrete Class
    void draw() { Shape::draw(); cout << "Triangle: Draw by Lines" << endl; }
};
class Quadrilateral : public Polygon { public: // Concrete Class
    void draw() { Shape::draw(); cout << "Quadrilateral: Draw by Lines" << endl; }
};
class Circle : public ClosedConvex { public: // Concrete Class
    void draw() { Shape::draw(); cout << "Circle: Draw by Brushtroke Algorithm" << endl; }
};
class Ellipse : public ClosedConvex { public: // Concrete Class
    void draw() { Shape::draw(); cout << "Ellipse: Draw by ..." << endl; }
};
int main() {
    Shape *arr[] = { new Triangle, new Quadrilateral, new Circle, new Ellipse };
    for (int i = 0; i < sizeof(arr) / sizeof(Shape *); ++i) arr[i]->draw();
    // ...
    return 0;
}
```

At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

Now, one more point that you should note is if I define a function - virtual function to be pure, then the class becomes abstract by definition. And you cannot create an instance, but the fact that a function is purely virtual says that an implementation for that function is not necessary, but it does not say that I cannot have an implementation. A function would be purely virtual, and in addition, I could have an implementation for that. Now if I provide the implementation, also its purity does not go away, because I am saying that it is a pure function. So, it will continue to remain pure, which means that no the class will continue to be abstract and no instance of that class can be created, but the function has an implementation which can be used. Now why would I need that, certainly the reason I will need that is for a code reuse for code factoring because a certainly the pure virtual function are at the root.

(Refer Slide Time: 26:50)

The slide title is "Shape Hierarchy: A Pure Virtual Function may have a body!". The code shown is:

```
#include <iostream>
using namespace std;
class Shape { public:
    virtual void draw() = 0; // Pure Virtual Function
    { cout << "Shapes: Init Brush" << endl; }
};
class Polygon : public Shape {
    void draw() { Shape::draw(); cout << "Polygon: Draw by Triangulation" << endl; }
};
class ClosedConvex : public Shape {
    // draw() inherited - Pure Virtual
};
class Triangle : public Polygon { public:
    void draw() { Shape::draw(); cout << "Triangle: Draw by Lines" << endl; }
};
class Quadrilateral : public Polygon { public:
    void draw() { Shape::draw(); cout << "Quadrilateral: Draw by Lines" << endl; }
};
class Circle : public ClosedConvex { public:
    void draw() { Shape::draw(); cout << "Circle: Draw by Brushtroke Algorithm" << endl; }
};
class Ellipse : public ClosedConvex { public:
    void draw() { Shape::draw(); cout << "Ellipse: Draw by ..." << endl; }
};
int main() {
    Shape *arr[] = { new Triangle, new Quadrilateral, new Circle, new Ellipse };
    for (int i = 0; i < sizeof(arr) / sizeof(Shape *); ++i) arr[i]->draw();
    // ...
    return 0;
}
```

The code is annotated with arrows pointing from the text "there is one part is a triangle algorithm which comes here" to the "Triangle" class definition, and from the text "all of these will be a common part of code which goes into all of this different draw functions" to the "Shape" class definition.

So, think about this; suppose in the first case, I needed to do a draw() for a triangle. Now if I need to do a draw() triangle, then certainly if you think about the drop, now there is there is one part is a triangle algorithm which comes here. But to be able to draw a triangle, I need to pick up a brush, I need to decide the color, I need to decide you know the thickness of the brush, I need to know the position, and all those. Now that part of the algorithm does it change between whether I am drawing a triangle or I am drawing a quadrilateral, drawing a quadrilateral also I need to pick a brush, I need to pick a color, I need to pick a thickness. In drawing a circle also, I will need to do that; in drawing an ellipse also, I need to do that. So, all of these will be a common part of code which goes into all of this different draw functions. What would be best is I take that common part of the code out and put it in the root.

So, what I simply need to do is when I am implementing draw function for the triangle, I simply invoke the draw function of shape. We have already seen this in earlier example, in the print example we saw this that you can always call the inheritable function form your parent directly by using the class of a parent. I could have made it propagate through polygon also that is I could have done it that this actually is a polygon :: draw(). So that it calls this and then that calls shape it could have been like that, but for the draw I did not find any reasonable logic as to what can triangle and quadrilateral share in

common form the draw of the polygon function, which actually has to draw. But certainly it has a lot that it can share from the draw of the shape function which can do all the required brush initialization and stuff like that before drawing a particular geometric object.

So, it is possible that pure virtual functions may have a body, and that does not take away the virtuality of it. So, pure virtual functions, I would am particularly emphasizing because I have seen quite a few textbooks, do not clarify this point. They just say that the pure virtual functions do not have a body, but I am just clarifying that it is not the pure virtual functions do not have a body; pure virtual functions may not have a body. They can if they want, but the purity lies in terms of whether you define it has a pure and that will lead the corresponding class to be an abstract base class.

(Refer Slide Time: 29:29)

The screenshot shows a presentation slide titled "Shape Hierarchy". The slide content includes:

- A code snippet for `main()` that creates an array of `Shapes` and calls their `draw()` methods.
- An "Output:" section showing the actual console output of the program, which lists the draw methods for each shape type: `Shapes: Init Brush`, `Triangle: Draw by Lines`, `Shapes: Init Brush`, `Quadrilateral: Draw by Lines`, `Shapes: Init Brush`, `Circle: Draw by Bresenham Algorithm`, `Shapes: Init Brush`, and `Ellipses: Draw by ...`.
- A bullet point stating: "Instances for class Shapes and class ClosedConics cannot be created".

So, if I do this then naturally my earlier output will change. And now for the same code, I have I am basically when I am here with index 0 - the `arr[0] → draw()` is actually calling for this function, which is calling the draw function of triangle which in turn is calling the draw function of shape. So, these two, first the draw function of shape executes and then that of because it has been called that way, it is not by structure. Similarly, for quadrilateral, similarly for circle, for ellipse, so every time you first call this. So, this is

one of the though it is not the only way to use the implementation of a polymorphic function of a pure virtual function, but this is one of the ways where you might use the implementation for a pure virtual function as well. But of course, that does not change the abstract notion of the classes, and shape and class closed conics continue to be abstract, and no instances can be created for them.

(Refer Slide Time: 30:35)

The slide is titled "Module Summary" and features a blue header bar with the title. On the left, there is a sidebar with a logo and a list of topics: Module 28, Partha Pratim Das, Objectives & Outline, Virtual Destructor, Pure Virtual Function, Abstract Base Class, and Summary. Below the sidebar is a circular portrait of a man. The main content area contains a bulleted list: • Discussed why destructors must be virtual • Introduced Pure Virtual Functions • Introduced Abstract Base Class. At the bottom, the footer includes the text "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and the number "14".

So, to conclude, we have discussed about why destructors must be virtual. And to be able to work on the polymorphic hierarchy, we have introduced pure virtual functions and introduced the notion of abstract base class. In the next module, we will take up more examples to show how these tools can be used to actually do certain design and code processing on the class hierarchy.

Programming in C++
Prof. Partha Pratim Das
Department Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 44
Dynamic Binding (Polymorphism): Part IV

Welcome to module 29 of Programming in C++. Since the last three modules, we have been discussing about polymorphism, the dynamic binding feature of C++. We have seen how on a hierarchy of classes, we can have virtual functions and how virtual functions dynamically bind to the object instance instead of binding to the pointer or reference to which a particular virtual function is being invoked. We have seen the differences between the semantics of non - virtual functions and virtual functions; you have seen the behaviour under overloading.

We have also noted that virtual functions could be pure and in which case they do not need to have a body and in those cases the corresponding class containing such pure virtual functions are abstract base classes. So, we have more or less covered the whole of the theory of polymorphism that is involved in C++. In this module and the next we will try to use the knowledge that we have acquired in terms of the polymorphic feature and engage that in terms of solving problems and doing sample design. So, in view of this, in the current module and also the next the target would be to understand design with class hierarchy.

(Refer slide Time: 01:54)

The screenshot shows a presentation slide titled "Module Outline". The slide has a dark blue header bar with the title "Module Outline" and a standard Windows-style toolbar above it. On the left, there is a vertical sidebar with a logo at the top, followed by the text "Module 29" and "Partha Pratim Das". Below this, under "Objectives & Outline", are several items: "Binding Exercise", "Staff Salary Processing", "C Solution", "C++ Solution", "Non-Polyorphic Hierarchy", "Polymorphic Hierarchy", and "Polymorphic Hierarchy (Flexible)". At the bottom of the sidebar, there is a link to "NPTEL MOOCs Programming in C++". The main content area of the slide is currently empty. At the very bottom, there is a thin black footer bar with the text "NPTEL MOOCs Programming in C++" on the left, "Partha Pratim Das" in the center, and the number "3" on the right.

Specifically, in this module, we will first take up and exercise with variety of binding combinations, when different kind of static and dynamic binding get mixed along with overloading how does it work. So, we will take one more little bit involved example to clear out the ideas and then will start off with an example of designing a staff salary application, which will run into the next module also. In the current module, you will show in how such an application can be designed in C and we will show the refinements C++ finally, using the polymorphic hierarchy in the next module.

(Refer slide Time: 02:44)

The slide is titled "Binding: Exercise". On the left, there is a sidebar with a logo, the text "Module 29", "Partha Pratim Das", "Objectives & Outline", "Binding: Exercise", "Staff Salary Processing C Solution Summary", and a small portrait of Partha Pratim Das.

The main content area contains two columns of code:

```
// Class Definitions
class A { public:
    virtual void f(int) { }
    virtual void g(double) { }
    int h(A *) { }
};
class B: public A { public:
    void f(int) { }
    virtual int h(B *) { }
};
class C: public B { public:
    void g(double) { }
    int h(B *) { }
};
```

// Application Codes

```
A a;
B b;
C c;
```

```
A *pa;
B *pb;
```

Below the code is a table titled "Initialization" with four columns:

Invocation	pa = &a;	pa = &b;	pa = &c;
pA->f(2);	A::f	B::f	B::f
pA->g(3.2);	A::g	A::g	C::g
pA->h(&a);	A::h	A::h	A::h
pA->h(&b);	A::h	A::h	A::h

At the bottom of the slide, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

The outline will be visible on the left of the slide as always. So, let us get started first with an example. So, in this example we have a simple structure. We have three classes; A is a base class, B is a specialization of A and C is a specialization of B. So, it is simple in that way. There are three functions; f(), g() and h(). In class A of which, f() and g() are defined to be virtual. So, they are polymorphic and we have a class H which is non virtual. In class B, we have f() and h() which mean that we are overriding f() and overriding h(), but we are simply inheriting g(). So, B does not have a signature of g() and while we override h we do two things, one is we change the parameter type from A * to B * and also make it a polymorphic function. In C further, we override g() and h() from whatever C has inherited and just use inherited version of the function f() in B in terms of C. So, this is the basic structure, we have constructed multiple objects a ,b, c and we have two pointers; one pointer to A another is pointer to B.

In this context, what we need to answer is, if we have these four invocations that is pa this pointer invokes function f with the parameter, function g with a parameter, function h with two different parameters. If we look at these four invocations and if we try to see on to the different columns where pa, this pointer has been set with a certain objects address, so here it is set with the address of an A object, here of B object and here of C object the task is to fill up this matrix, which is already filled up here. So, what I will do I

will just quickly run you through as to why we should see the invocation the bindings in way that they are. So, to start with, we start with pa being assigned &a which say that for pa in the static type is A* as we can easily see and also the dynamic type this is an a type object. So, the dynamic type of pa is also A.

So, since the static type is pa naturally it looks into the class A for the binding and the dynamic type being A clearly tell us that certainly, in all of this cases all these will actually invoke the class A functions. The only point to note here that in terms of the h function which takes A* pointer, whereas in the third case we have actually invoke it with the A* pointer, but in the last case we have invoked it with the B* pointer. So, if you look into this then this is pa invoking h() with B*.

Now, we already know that B ISA A, and we already know that upcast is possible. So, this will also be possible because what will happen is this B* pointer; will get up cast to the A* pointer and the function will get invoked and that is the reason you have the same h function being invoked in both these cases. So, does not something very interesting let us look at the next one, the next is where pa has static type of A and the dynamic type of B because we have b object. So, naturally again pa being statically of type A, it will always look at this. So, now when you do pa → f(), it looks at the function the class A, and then it finds that the function f is virtual, which means that it needs to be dynamically bound. It will be bound by the actual object that pa is pointing to and that object is B type and B overrides this function. So, earlier case it had invoked A :: f(), Now, it will invoke B :: f() which is straight forward.

In the second case again, when we look at pa invoking g(), it invokes A :: g() because even though g is virtual B does not override g. There is no g in class B. So, actually because it is virtual, this is actually trying to invoke B :: g(), but B :: g() is same as A :: g() because b has simply inherited the function g. So, you have A :: g() here. In last two cases, you do not have anything further because function h is in class A is non virtual. So, it is bound by the type of the pointer.

Let us move to the third, where I have an object of type C and I have pointer in to that. Naturally, if again the first place to look at is class A and we are invoking function f. So,

f is virtual, which means that it will get dispatched to class C, the type of the actually pointed object. So, if it gets invoked to in dispatch to class C then basically we should be invoking C :: f(), but C does not override f(). So, C has simply inherited the f from B that is why you get B :: f() here. Whereas, when you try to do invoke function g then again you come here and you have a virtual function so that gets dispatch to certainly C :: g() because you have C object. So, you have C :: g() in this case C has over written whatever it had inherited. So, it has a separate C :: g() function. So, that gets invoked and finally, in terms of the h function again you have the same behaviour because they are not virtual functions.

So, this how it behaves if these three objects, I invoke methods for these three objects using their address as if as a and a pointer to the a type of address. So, next what we look at what happens if we do the same way try to do the same invocation using the B type of pointer.

(Refer slide Time: 10:07)

Binding: Exercise

```

Module 29
Partha Pratim Das
Objectives & Outcomes
Binding Exercise
Staff Salary Processing
C Solution Summary

```

```

// Class Definitions
class A { public:
    virtual void f(int) { }
    virtual void g(double) { }
    int h(A *) { }
};
class B: public A { public:
    void f(int) { }
    virtual int h(B *) { }
};
class C: public B { public:
    void g(double) { }
    int h(B *) { }
};

```

```

// Application Codes
A a;
B b;
C c;
A *pa;
B *pb;

```

Initialization			
Invocation	pb = &a;	pb = &b;	pb = &c;
pB->f(2);	Error	B::f	B::f
pB->g(3.2);	Downcast	A::g	C::g
pB->h(&a);	(A *) to (B *)	No conversion (A *) to (B *)	
pB->h(&b);	(B *)	B::h	C::h

NPTEL MOOCs Programming in C++ Partha Pratim Das S

So, that is in the next slide. So, we are in this, there is no change on this side, what has changed is the invoke initialization the pointer b. Now, which we are using to invoke, earlier we were doing in terms of pa now, we are doing in terms of pb. Now, certainly the first thing that we see is this is a pb points to a. So, what do we expect? What is the static

type here? The static type is B^* , and what is the dynamic type here is A^* . So, if this assignment has to happen, if this assignment has to take place, then some A^* object as to get cast to B^* which happens to be a downcast. So, this down cast is not permitted. So, in all of these cases, actually we will not be able to do anything at all because this particular assignment itself will error out, because it is an error in terms of down cast. So, these cases are over.

Let us move to next one, which is using this and when we point to a b object. So, both the static and dynamic type is B . So, as we do that as in the previous case if I do $pb \rightarrow f()$ then I start actually here because the type of the pointer is b here if should be invoked and f is a virtual function as we have seen it is a virtual function and it has a dynamic and it is pointing to a b object. So, it should invoke the function of the B object. Similarly, if we invoke g it should again start here and try to invoke in a similar manner $B :: g()$ because g is a virtual function again, but B has not overridden g . So, B has simply inherited the g function from A . So, you see that $A :: g()$ will be invoked.

Let us look at the next one where we are trying to invoke this function. Now, when we try to invoke this function what will happen we are here? So, we start looking in the class B and in class B do I have a function h which is actually a function which is overridden from the function in A . Therefore, when I try to do $pb \rightarrow h()$ and if I pass $\&a$ then the type of the actual parameter is A^* , whereas the type of the formal parameter, what it expects is B^* . So, it means that I need to perform a conversion from A^* to B^* , only then I will be able to do that call, but certainly as we know that this turns out to be a down cast, this is a case of a downcast. So, this is not permitted. So, this particular function invocation will simply not compile.

Coming to the last, when you invoke h in terms of this pb and you try to see it will turn out to be $B :: h()$ because you certainly look here, you have B type object. So, certainly you will have $B :: h()$. Finally, when if you have the c object, this is these cases prevail because you start here and this function f is a virtual therefore, it gets relegated dispatch to $C :: f()$ and C as not overridden f . So, it what actually invokes is the f that C inherits, which is $B :: f()$, so that part is easy. Now, if you invoke function g you again start here now in this what happens you do not see a g but what is a g that you see here that is

actually this g, which is which B has inherited from the class A and that function is a virtual. So, actually B :: g() the function g in class B is same as function g in class A and that actually is a virtual function.

So, this will delegate according to the type of the object that dynamically exist. So, it will try to listen through this and it will try to invoke C :: g(), that is this function so you get C :: g(). The next case is a similar it will actually since this function as become virtual so you start looking in here. So, you see that h is a virtual function. So, you delegate dispatch that to the class C because you have C type object. So, you try to invoke this function and then you find that the actual parameter type is A* and the formal parameter type is B*. So, the conversation is a downcast. So, it here again you get and compilation error on this code. Finally, when you try to invoke h() certainly you have pb. So, you start here you find that h is a virtual function. So, you dispatch according to the type of the dynamic object which brings you to this and you have C :: h().

I have quickly run it through this, maybe it was little fast for some of you. If it is then please spend some more time try to understand re run the video understand the logic in every case, but this kind of I have tried to cover all different cases of static and non static binding along with the what can be overloaded and what can be cast in terms of the upcast and the downcast. So, with this we will now move on.

(Refer slide Time: 16:20)

Module 29
Partha Pratim Das
Objectives & Outline
Reading Material
Staff Salary Processing
E-Solutions
Summary

NPTEL MOOCs Programming in C++
Partha Pratim Das

- An organization needs to develop a salary processing application for its staff
- At present it has an engineering division only where **Engineers** and **Managers** work. Every **Engineer** reports to some **Manager**. Every **Manager** can also work like an **Engineer**
- The logic for processing salary for **Engineers** and **Managers** are different as they have different salary heads
- In future, it may add **Directors** to the team. Then every **Manager** will report to some **Director**. Every **Director** could also work like a **Manager**
- The logic for processing salary for **Directors** will also be distinct
- Further, in future it may open other divisions, like Sales division, and expand the workforce
- **Make a suitable extensible design**

With this we will take up a problem which we will try to solve in the remaining of this module and the next it is a staff salary processing problem. So, let us think of an organization that needs to develop a salary processing application for a staff. So, what is the information that we have we have that the organization has only one engineering division, where engineers and managers work every engineer reports to a manager and a manager can also work as a engineer. But what is different is the salary processing logic for the engineer and for the managers are different, possibly managers have some bonus and all that. So, the same function cannot be used to process the salary for both them.

Further what the organization wants that foresee that in some near future they will also possibly appoint directors in this division and then some managers will report to different directors and directors will would, if required would also work as managers and so on and directors will have their own processing logic for their salary. Further down in future, the organization wants to say that further down in future which is not in their view right now, but they are they could also open other divisions like they could open a sales division and expand their work force in a completely different type. So, what is required is in this context we need to make a suitable extensible design, design so that we can as a required add new employee types and new processing logic for the processing of salary without making significant changes to the existing code or if possible not making any change at all in the application that finally process the salary, this is the basic target.

(Refer slide Time: 18:09)

C Solution:
Engineer + Manager

- How to represent Engineers and Managers?
 - struct
- How to initialize objects?
 - Initialization functions
- How to have a collection of mixed objects?
 - Array of union
- How to model variations in salary processing algorithms?
 - struct-specific functions
- How to invoke the correct algorithm for a correct employee type?
 - Function switch
 - Function pointers

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, let us get started. So, I will take the clue from a solution which we will do in C and let us assume that we just have C at hand and let say how would have solve this problem assuming the very first version where we have just two types of employees; engineers and manager. So, there are several questions that you will need to answer before you can get started with the design and actually code, for example, how do you represent these concepts of engineers and managers, they will have to be represented somehow. So, possible since you are in C then the choices is almost trivial that you will use the struct that is not a problem. How do you initialize these objects with their name, their designation, their basic pay and so on? Certainly, you will need to have certain initialization functions that work for every structure type.

Third is a deeper question is finally, see if we have structures then we will have some structure for engineer and some structure for managers. These are two different kinds of structure types. So, if we have that then certainly we cannot make an array of engineers and managers. So, we will need a container because we can just make of an array of engineers or make an array of managers, but if you do that then the processing will get fragmented every time we add a new kind of employee. So, we need to have a container which can have any kind of objects that we can put together.

So, the solution in C is basically using array of union. So, what we can do we can say that I have a struct which has a fixed field, say type and then within that it has a union which has multiple different fields, say this is a pointer to engineer, this is a pointer to manager like this. So, what happens is when you want to keep that, you when you want to keep an engineer record you put a specific type value here, say the type value is Er and then set this pointer which is meant for to point to engineer object make this point to the particular instance of struct, but when you have to say do with deal with the manager then you set this type as say some manager value, manager type and set this pointer to the manager instance. Then depending on any element in this if you first check the type whether it is Er or it is Mgr you can decide which of the fields in the union to use and use it appropriately. So, this is a nice design wrapper which is C uses extensively to keep a container of objects which are of mixed types. So, we will have to use that.

Now, the certainly the question of how do we have different salary processing, this is more like the initialization. So, which we will need to have for every structure type, we will need to have some structure of a specific function a function for engineer a function for manager and so on. Then finally, comes the question that if we have an array of a union. So, we have different records, this record could mean for an engineer, this record could be for manager, this could be for an engineer, this could be again for an engineer and so on, this could be for manager and so on.

How do I given any object, how do I decide what is a correct algorithm for the correct employee type. So, there is some kind of a switching which is involved that is I have to know because the correct employee the correct algorithm is encoded in tempt of the structure specific function. So, I have and in the array of union I have the employee type. So, I have to combine them in some way and create a switch and C provides typically two different options that I can have a function switch that is if else if else kind of structure or I can use a set of function pointers.

(Refer slide Time: 22:25)

```
#include <stdio.h>
#include <string.h>

typedef enum E_TYPE { Er, Mgr } E_TYPE;

typedef struct Engineer { char *name; } Engineer;
Engineer *InitEngineer(const char *name) { Engineer *e = (Engineer *)malloc(sizeof(Engineer));
    e->name = strdup(name); return e;
}

void ProcessSalaryEngineer(Engineer **e) {
    printf("%s: Process Salary for Engineer\n", e->name);
}

typedef struct Manager { char *name; Engineer *reports[10]; } Manager;
Manager *InitManager(const char *name) { Manager *m = (Manager *)malloc(sizeof(Manager));
    m->name = strdup(name); return m;
}

void ProcessSalaryManager(Manager **m) {
    printf("%s: Process Salary for Manager\n", m->name);
}

typedef struct Staff { E_TYPE type;
    union { Engineer *pE; Manager *pM; } p;
} Staff;
```

So, let us start and look at that we will initially use a function switch. So, just to quickly run through that I need to define the types of employees that I have, Er for engineer; Mgr. So, I defining a enumerate type these are structure which defines and engineer type I had just shown name any other attributes can be put in there. We have structures for manager where we have name and I have also optionally put that since engineers will report to managers. I will need to keep a list of engineers who report to the manager we need to initialize.

So, there is an initialization function which certainly takes the name and then allocates the space for that particular structure object through malloc, sets the name by copying and returns that. So, if I invoke the init engineer function with a name then it will return me the pointer to an engineer which I can then put in the store. Similarly, for the manager I have init function like this. We have separate processing function for engineer, which takes an engineer pointer and manager which takes a manager pointer and finally, these are collection store as I said.

So, I have a structure where one field is a type which will be one of these because it is E_TYPE and I have a union of two pointers. So, at any point of time any one of them will actually carry a meaningful value the other one will because they will be overlap because

it is a union both of them do not exist at the same time. So, finally, I call this type as staff. So, that turns out to be my final collection of employees.

(Refer slide Time: 24:13)

The screenshot shows a presentation slide titled "C Solution: Engineer + Manager". On the left, there's a sidebar with navigation links: Module 29, Partha Pratim Das, Objectives & Outline, Reading Exercise, Staff Salary Processing, C Solution Summary, and a photo of the speaker. The main content area contains C code for processing staff salaries. The code initializes an array of staff records, each containing a type field (Er or Mgr) and a pointer to either an Engineer or Manager structure. It then iterates through the array, calling the appropriate processing function based on the type. The output section shows the names of the staff members followed by their respective salary processing messages. The footer indicates the slide is from NPTEL MOOCs Programming in C++.

```
int main() {
    Staff allstaff[10];
    allstaff[0].type_ = Er;
    allstaff[0].pE = InitEngineer("Rohit");
    allstaff[1].type_ = Mgr;
    allstaff[1].pE = InitManager("Kamala");
    allstaff[2].type_ = Mgr;
    allstaff[2].pE = InitManager("Rajib");
    allstaff[3].type_ = Er;
    allstaff[3].pE = InitEngineer("Karita");
    allstaff[4].type_ = Er;
    allstaff[4].pE = InitEngineer("Bhanubhu");

    for (int i = 0; i < 6; ++i) {
        E_TYPE t = allstaff[i].type_;
        if (t == Er) ProcessSalaryEngineer(allstaff[i].pE);
        else if (t == Mgr) ProcessSalaryManager(allstaff[i].pE);
        else printf("Invalid Staff Type\n");
    }
    return 0;
}
```

Output:

```
Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Karita: Process Salary for Engineer
Bhanubhu: Process Salary for Engineer
```

So, given this now I can certainly try to write the application. So, staff is my collection. So, I create an array of this union type. So, 10 staff can be kept in this all staff record then the next, this is just for making the whole program work. It just showed that how we would create the different engineer and manager and put them in the collection. So, if we just think about one in the first one then what we are trying to do we have this. So, this is allstaff[0], this as a type field.

So, there I put the engineer type because I want an engineer and then I invoke init engineer and as you saw that if I invoke it with the name of the engineer then you returns me pointer to the engineer. So, in the union I am basically looking at a engineer pointer which will get set. In this way, the manager pointer and the other engineer pointers also get set having done this then this is where I actually do the actual processing since what I will have to do; I will go over this whole collection. So, I have to go over this whole collection. Here, I have just hard coded the number 6 ignore that this could have been tracked also by counting. I just hard coded that there are 6 or possibly that it should have been 5. There are 5 such and so what I do is I have to go over.

So, I have an array of all this union records. So, I go to the 0th one, first I look at the typefield. So, because I have to know which kind of, whether I have an engineer or a manager. So, you pick up the type here from allstaff [i] the type and then depending on the type if the type is Er, I invoke the processing salary engineering function with the corresponding pointer which is the pE pointer, if not otherwise I check if it is a manager type then I invoke the process salary manager function with the corresponding manager pointer(pM) and if it is even not even that then there must be some error. So, I just error print a error message.

If I do that then we have these 5 employees, 1 engineer and 3 engineer and 2 managers will get this particular output in terms of processing. So, this output is equivalent to as if the salary has been processed. So, this is how this could be written. So, naturally you can see there are some annoying points, one is a union container itself, one is a way you have to initialize and create this object and one certainly is this. So, if we if we just go forward and try to now take the first step in the future that you want add the Director type here or design remain same. So, all that all this design details remain same, but only thing we are adding only thing we are adding a director type now. So, we want to add that.

(Refer slide Time: 27:17).

```

Module 29
Partha Pratim Das
Objectives & Outcome
Holding Exercise
Staff Salary Processing
C Solution
Summary

C Solution:
Engineer + Manager + Director

#include <stdio.h>
#include <string.h>

typedef union E_TYPE { Eng, Mgr, Dir } E_TYPE;

typedef struct Engineer { char *name; } Engineer;
Engineer *InitEngineer(const char *name) { Engineer *e = (Engineer *)malloc(sizeof(Engineer));
    e->name = strdup(name); return e;
}
void ProcessSalaryEngineer(Engineer *e) {
    printf("%s: Process Salary for Engineer\n", e->name);
}

typedef struct Manager { char *name; Engineer *reports; [10]; } Manager;
Manager *InitManager(const char *name) { Manager *m = (Manager *)malloc(sizeof(Manager));
    m->name = strdup(name); return m;
}
void ProcessSalaryManager(Manager *m) {
    printf("%s: Process Salary for Manager\n", m->name);
}

typedef struct Director { char *name; Manager *reports; [10]; } Director;
Director *InitDirector(const char *name) { Director *d = (Director *)malloc(sizeof(Director));
    d->name = strdup(name); return d;
}
void ProcessSalaryDirector(Director *d) {
    printf("%s: Process Salary for Director\n", d->name);
}

typedef struct Staff { E_TYPE type; union { Engineer *pEng; Manager *pMng; Director *pDir; } Staff;
} Staff;

```

NPTEL MOOCs Programming in C++ Partha Pratim Das 11

So, if we do that then all that it means that I have to add a structure type for the director. The initializer for director, the processing routine for the director in this there will have to be a code for the director. They will have to be another field which points to the director type. So, that would basically be all in terms of the representation.

(Refer slide Time: 27:40)

```

Module 29
Partha Pratim Das
Objectives & Outcomes
Handing Exercise
Staff Salary Processing
C Solution Summary

C Solution:
Engineer + Manager + Director

int main() {
    Staff a138staff[10];
    a138staff[0].type_ = Er;
    a138staff[0].p0 = InitEngineer("Rohit");
    a138staff[1].type_ = Mgr;
    a138staff[1].p0 = InitManager("Kamala");
    a138staff[2].type_ = Mgr;
    a138staff[2].p0 = InitManager("Rejik");
    a138staff[3].type_ = Er;
    a138staff[3].p0 = InitEngineer("Kavita");
    a138staff[4].type_ = Er;
    a138staff[4].p0 = InitEngineer("Shambhu");
    a138staff[5].type_ = Dir;
    a138staff[5].p0 = InitDirector("Banjana");

    for (int i = 0; i < 6; ++i) {
        E_TYPE t = a138staff[i].type_;
        if (t == Er) ProcessSalaryEngineer(a138staff[i].p0);
        else if (t == Mgr) ProcessSalaryManager(a138staff[i].p0);
        else if (t == Dir) ProcessSalaryDirector(a138staff[i].p0);
        else printf("Invalid Staff Type\n");
    }
    return 0;
}

Output:
Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rejik: Process Salary for Manager
Kavita: Process Salary for Engineer
Shambhu: Process Salary for Engineer
Banjana: Process Salary for Director
NPTEL MOOCs Programming in C++

```

Then, if we actually look at the code then all this were there, then I can also create a director and put that, but here if you look at then in the processing routing, if you look at then you will need to have another condition which will check if somebody is a director and then call the corresponding process salary director function with the pointer of the direct. So, if you just see, we are getting into some bit of uncomfortable situation here because more and more types the basic design specification it has to be flexible, but more and more we add different types, we on one side here, this switch will keep on expanding.

So, the application code if I add a type then the application code has to know this is the application code main routine right. So, the application code as is to know that a new type as been added and it as to add some code there it as to add some more switch there. So, that is one a knowing factor we having to maintain the type information through an enumerated value separately, which is another disturbing factor and if you just look at if

you just look at the earlier one, we are creating lot of duplications, for example, here we have to do a union based collection which is also quite cumbersome because it is not guarantee that the value that I keep in the type and the particular field that I read in the union is no way checked by compiler.

So, it we can always make mistakes in that and we see lot of issues because see this name see if you look into this typedef this part in the director. This part all of these are basically code you know repeated things and if you look into these codes all these codes the copy of name and all that they are very similar in structure, but still we are just having to copy and you know paste and edit into that and creating lot of possible issues in terms of the code maintenance and expansion. So, by using C we are able to get to a solution we can survive, but we need to have better solutions for this problem, which is what we will discuss in the next module.

(Refer slide Time: 30:02)

Module Summary

- Practiced exercise with binding – various mixed cases
- Started designing for a staff salary problem and worked out C solutions

NPTEL MOOCs Programming in C++ Partha Pratim Das 13

So, here to summarise we have practiced a binding exercise and we have started designing for a staff salary application, where we have just worked out C solution and observed what are the difficulties lacunae in that solution.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 45
Dynamic Binding (Polymorphism): Part V

Welcome to Module 30 of Programming in C++. We have been discussing about Polymorphism in C++, after introducing the various features of polymorphism from the last module we have started working on solving a specific problem and we want to explore how polymorphism could be effective in that solution of the problem.

(Refer Slide Time: 00:45)

The screenshot shows a presentation slide with the following details:

- Module Objectives** (Title)
- Module 30** (Module Identifier)
- Partha Pratim Das** (Instructor Name)
- Objectives & Outline** (Section)
- Staff Salary Processing** (List item)
- C Selection** (List item)
- C++ Selection** (List item)
- Memory Management** (List item)
- Polymorphism** (List item)
- Hierarchy** (List item)
- Polymorphism Hierarchy** (List item)
- Summary** (List item)
- Understand design with class hierarchy** (List item)

At the bottom of the slide, there is footer text: "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and the number "2".

So the objective is to continue to understand the designing with class hierarchy.

(Refer Slide Time: 00:51)

This slide shows the 'Module Outline' for 'Staff Salary Processing'. The left sidebar lists 'Module 30' topics: Partha Pratim Das, Objectives & Outline, Staff Salary Processing (C Solution, C++ Solution, Non-Polymorphic Hierarchy, Polymorphic Hierarchy, Polymorphic Hierarchy (Flexible)), and Summary. A video thumbnail of a man speaking is also present. The main content area is titled 'Module Outline' and contains a bullet point: '• Staff Salary Processing' with sub-points: '• C Solution', '• C++ Solution', '• Non-Polymorphic Hierarchy', '• Polymorphic Hierarchy', and '• Polymorphic Hierarchy (Flexible)'. The footer includes 'NPTEL MOOCs Programming in C++', 'Partha Pratim Das', and a page number '3'.

And the specific outline would be we had discuss the C solution in the last module we will discuss the C++ solution, and we will discuss three versions of the C++ solutions one after the other and you will see that on the left hand side of your slides.

(Refer Slide Time: 01:08)

This slide is titled 'Staff Salary Processing: Problem Statement: RECAP (Module 29)'. The left sidebar is identical to the previous slide. The main content area contains a bulleted list of requirements: '• An organization needs to develop a salary processing application for its staff', '• At present it has an engineering division only where Engineers and Managers work. Every Engineer reports to some Manager. Every Manager can also work like an Engineer', '• The logic for processing salary for Engineers and Managers are different as they have different salary heads', '• In future, it may add Directors to the team. Then every Manager will report to some Director. Every Director could also work like a Manager', '• The logic for processing salary for Directors will also be distinct', '• Further, in future it may open other divisions, like Sales division, and expand the workforce', and '• Make a suitable extensible design'. The footer includes 'NPTEL MOOCs Programming in C++', 'Partha Pratim Das', and a page number '4'.

Just a quickly recap, this is the staff salary processing problem and organization needs to process the salary they have one division with engineers and managers, managers can also behave as engineers, there a separate logic for processing salary for engineer as

well as manager. In future the company intends to add the position of directors in the same division who will be able to also work as managers and will have a separate salary processing logic for them which is different from the engineers as well as manager. And further down in future the company wants to keep it open that they can add other divisions like the sales division and have all whole lot of different kinds of staff there.

So, in view of this the whole challenge is to how we make a suitable extensible flexible design for the salary processing.

(Refer Slide Time: 02:04)

The screenshot shows a presentation slide with the following details:

- Title:** C Solution: Engineer + Manager: RECAP (Module 29)
- Module 30** (in the top left corner)
- Partha Pratim Das** (in the top left corner)
- Objectives & Outline** (link in the top left corner)
- Staff Salary Processing** (link in the top left corner)
- C Solution** (link in the top left corner)
- Module 29** (link in the top left corner)
- Partha Pratim Das** (link in the top left corner)
- Summary** (link in the top left corner)
- Partha Pratim Das** (in the bottom right corner)
- NPTEL MOOCs Programming in C-I-I** (in the bottom left corner)
- 5** (in the bottom right corner)
- Content:** A bulleted list of topics:
 - How to represent Engineers and Managers?
 - struct
 - How to initialize objects?
 - Initialization functions
 - How to have a collection of mixed objects?
 - Array of union
 - How to model variations in salary processing algorithms?
 - struct-specific functions
 - How to invoke the correct algorithm for a correct employee type?
 - Function switch
 - Function pointers

We started off with a C solution, and to attack the solution we identified some core questions that need to be answered to make a successful design and these are the C answers and we have already seen the solution in terms of the C answers. We have seen how the code will look like if I have just the engineer and manager and how what changes will be required when I add the directors and we saw different lacunae (Refer Time: 02:33) in terms of the (Refer Time: 02:34).

(Refer Slide Time: 02:33)

The slide title is "C++ Solution: Non-Polymorphic Hierarchy Engineer + Manager". It features a class hierarchy diagram with "Manager" pointing to "Engineer". Below the diagram is a bulleted list of questions:

- How to represent Engineers and Managers?
 - Non-Polymorphic class hierarchy
- How to initialize objects?
 - Constructor / Destructor
- How to have a collection of mixed objects?
 - array of base class pointers
- How to model variations in salary processing algorithms?
 - Member functions
- How to invoke the correct algorithm for a correct employee type?
 - Function switch
 - Function pointers

The sidebar on the left lists topics: Module 30, Partha Pratim Das, Objectives & Outcome, Staff Salary Processing, C Solution, C++ Solutions, Non-Polymorphic Hierarchy, Polymorphic Hierarchy, Polymorphic Hierarchy (Details), Summary. The footer includes NPTEL MOOCs Programming in C++, Partha Pratim Das, and a page number.

Now move on to C++ solution and, we start by first modeling that on the fact that the company has said a manager can behave as a engineer as well. So, based on that we first start by making a model saying that a manager ISA engineer, so that is a hierarchy beginning of the hierarchy. Since we have just starting off we do not know how it will take shape. Earlier just extract we are now saying that it is a non-polymorphic class hierarchy, this is a basic class hierarchy which has to be there. But the movement we come to C++ several solutions look different, the initialization etc.. become constructor and destructor. We do not need that for the collective container some problems are automatically getting solved because we have come to C++.

We do not for example need the wrapper of union and the collection, I mean array of unions in terms of keeping the collection of next objects we can just use the. Since if this is this is the base class because manager ISA engineer, so we can just keep an array of the pointers to base class and keep the objects their and what will be using in this is a future that the derive class pointer can be cast as a base class pointer without any loss of information in terms of the (Refer Time: 04:05). So, this becomes our container now.

Certainly we had differently named functions for different structure types, structure put specific functions. Now in the class we will have member functions to do the job, and in

terms of our dispatch we will continue to use the functions switch or the function pointers. This is our basic solution premise that we would like to see.

(Refer Slide Time: 04:31)

```

Module 30
Partha Pratim Das
Objectives & Outline
Staff Salary Processing
C Solution
Non-Polymorphic Hierarchy
Non-Polymorphic Hierarchy (Details)
Summary
#include <iostream>
#include <string>
using namespace std;

typedef enum E_TYPE { Er, Mgr } ;
class Engineer : protected string name_, E_TYPE type_ {
public: Engineer(const string& name, E_TYPE e = Er) : name_(name), type_(e) {}
    E_TYPE GetType() { return type_; }
    void ProcessSalary() { cout << name_ << " " << "Process Salary for Engineer" << endl; }
};

class Manager : public Engineer { Engineer *reports_[10];
public: Manager(const string& name, E_TYPE e = Mgr) : Engineer(name, e) {}
    void ProcessSalary() { cout << name_ << " " << "Process Salary for Manager" << endl; }
};

int main() { Engineer e1("Rohit"), e2("Kavita"), e3("Shambhu");
    Manager m1("Ranjan"), m2("Ranjib");
    Engineer *staff[] = { &e1, &e1, &e2, &e2, &m3 };
    for (int i = 0; i < sizeof(staff) / sizeof(Engineer*); ++i) {
        E_TYPE t = staff[i]->GetType();
        if (t == Er) staff[i]->ProcessSalary();
        else if (t == Mgr) (Manager *)staff[i]->ProcessSalary();
        else cout << "Invalid Staff Type" << endl;
    }
    return 0;
}

```

Let me just quickly take you through what the code could look like. So, we continue to have this we have an enumerated type to remember the whether have an engineer or a manager will see why this is still required. What we change significantly is, we change from the structure to the class. So now you have a class, certainly the name data member becomes a protected member here, I have a type member which remembers that this is an engineer. Then I have a constructor which does a job of the initialization and it takes this name and type and sets it in these two variables.

Now, you will have to see that earlier in the C solution the structure did not have an E_TYPE variable, because you had maintaining that outside of the objects in terms of the part of the structure with the union that we created in the collection. Now we are getting rid of that so it has to be maintained within the object itself, which is kind of funny, which is kind of not a not a very healthy design because or the fact that it is an engineer object should be able to tell me that actually that the type is Er, but for the kind of non-polymorphic hierarchy we will possibly need this information. Certainly, besides the

constructor there is a way to get this type information and there is a method to process the salary.

We specialize from here, we go to the manager we has specialization of engineer we add new data members so that managers can actually keep the recording reporting information. The constructor for initialization the constructor has to use the constructor of engineer to actually initialize name. By this simple process of bringing it in terms of a hierarchy we are getting rid of lot of code duplication that we had earlier in every structure we needed to have a string name. Now we just have that in the base and through the initialization of the derive class invoking the constructor of the base class we can set this name directly from here. So here, it comes to this and then is set to the actual base class part.

We have also been able to get rid of differently named functions for different structures, because earlier it was process salary engineer process salary manager now you have process salary all that is happening is between these two there is a inheritance and overriding so we over write the process salary and now proved the logic of manager salary here.

So, tell these point things are pretty good and this is where we construct the objects because we have explicit constructor so we are just constructing the objective. We could have a dynamically constructed it also that does not add any special value. Then we say that we have an array of base pointers, so staff is a array of base pointers which is engineer and we put all these pointers there in the same order that we have done earlier. You just put the pointers to these objects. So this array holds whole the set of pointers, and so I can go for this.

Now, I come to the interesting part of how my application looks. My application again I have to iterate over this array which is the size, but my application again first has to know what is that type. Because you may if I access an array element, if I access `staff[i]` then I get an engineer pointer because that is a static type. But in actuality it could be a engineer object or it could be a manager object I do not know what it is, but unless I know that I do not know whether this function should be called or this function should be

called I have no idea. So I have to decide on that, I have to decide on which function I would need to call.

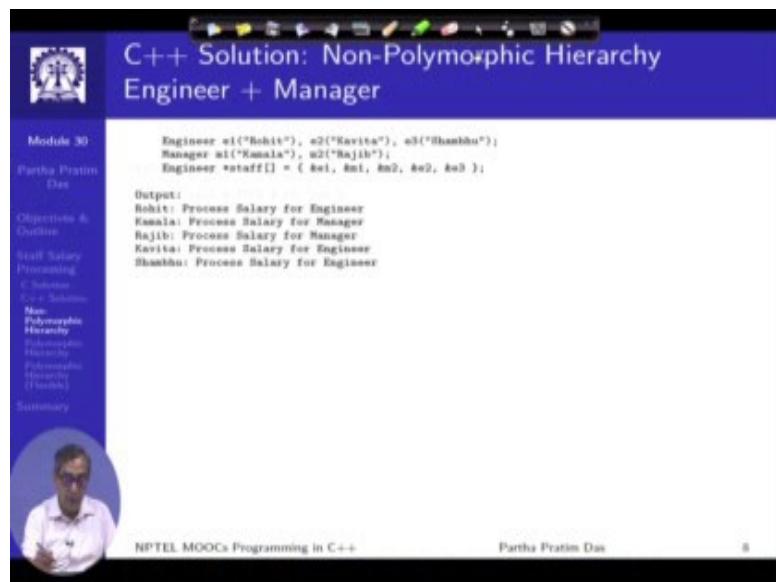
This is where the type information will be come handy. So, I take the type information I access it though they get type method in the base class the engineer class so this i get the type and then I compare whether it is. So, I am basically doing a function switch here, switch function as I did in case of C here, and if that does not match I will check for manager and so on. Once it matches, suppose it matches this then I can directly equal this because I know staff [i] is of type engineer. If this matches as Er then the type of object that I have is an engineer object, so I can directly invoke the process salary of the engineer class. But if it does not match then I go and check if it is a manager. If it is a manager so whether I have a manager objects which is what I will have if I go to the next one.

But now if this matches then I need to invoke this function, so how do I invoke this function? I need to cast the staff [i] into, I actually know by having access this information from the type I actually know that though the pointer is of engineer type I actually have a manager type. So, I take this pointer of which is engineer type and cast it to the manager type, I am trying to do something which is dangerous. Because what am I doing? I am doing a this is a engineer, this is manager, staff is a pointer here, staff [i] is a pointer here, pointed to this I am just trying to bring it down to be a pointer to manager. I am doing a downcast and I have done that cast in terms of the way C allows me to do the cast which is the forced cast.

Just recall if this has become hazy then I would suggest that you go back to our initial discussion on the casting and you will see that downcast can be forced in this way. So I force this. How could I confidently do that? I could confidently do that because I am managing that type. I know because when manager was created then certainly this type was passed on. So, it went here then it went here then it got set in the type field. So I know that it is manager type therefore I cast it to the manager type. Once I cast this pointer to be a manager type and then invoke process salary then certainly it invokes the process salary member function of the manager class.

Basically, this in this process though some of the issues that existed in the C design has been removed but still the dispatch process remains to be quit vulnerable; dispatch process depends on being able to manage this type so well, and it is further problematic because if I have to add one more type of employee then several code will have to change and particularly disturbingly the application code has to change.

(Refer Slide Time: 11:57)



The screenshot shows a presentation slide titled "C++ Solution: Non-Polymorphic Hierarchy" and "Engineer + Manager". The slide is part of "Module 30" by Partha Pratim Das. On the left, there's a sidebar with navigation links: "Objectives & Outline", "Staff Salary Processing", "C Solution", "C++ Solution", "Non-Polymorphic Hierarchy", "Polymorphic Hierarchy", "Polymorphic Hierarchy (Example)", and "Summary". A video thumbnail of a man speaking is also present on the left.

C++ Code:

```
Engineer *e1("Rohit"), *e2("Kavita"), *e3("Shambhu");
Manager *m1("Kamala"), *m2("Rajib");
Engineer *staff[] = { e1, m1, m2, e2, m3 };
```

Output:

```
Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Kavita: Process Salary for Engineer
Shambhu: Process Salary for Engineer
```

At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

So let us take a look, this is the output you can check later on that output is correct.

(Refer Slide Time: 12:02)

The slide title is "C++ Solution: Non-Polymorphic Hierarchy Engineer + Manager + Director". It features a navigation bar with icons for back, forward, search, and other controls. On the left, there's a sidebar with "Module 30" and "Partha Pratim Das" at the top, followed by a list of topics: Staff Salary Processing, C Solution, Non-Polymorphic Hierarchy, Polymorphic Hierarchy (Abstract), and Summary. Below the sidebar is a circular profile picture of Partha Pratim Das.

The main content area contains a class hierarchy diagram with three classes: Director, Manager, and Engineer, connected by arrows pointing from Director to Manager and Manager to Engineer. Below the diagram is a bulleted list of learning objectives:

- How to represent Engineers, Managers, and Directors?
 - Non-Polymorphic class hierarchy
- How to initialize objects?
 - Constructor / Destructor
- How to have a collection of mixed objects?
 - array of base class pointers
- How to model variations in salary processing algorithms?
 - Member functions
- How to invoke the correct algorithm for a correct employee type?
 - Function switch
 - Function pointers

At the bottom of the slide, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das". There is also a small number "9" in the bottom right corner.

So, let us move on and let us try to add the director the first step in the future. Director ISA manager which we have got the information. So you continue to use the non-polymorphic class hierarchy, rest of the design do not change so it just that the hierarchy has to extend.

(Refer Slide Time: 12:20)

The slide title is "C++ Solution: Non-Polymorphic Hierarchy Engineer + Manager + Director". The code shown defines three classes: Director, Manager, and Engineer, each with a ProcessSalary() member function that prints a specific message based on its type. The Director class inherits from Manager, and Manager inherits from Engineer. The main() function creates instances of all three types and processes their salaries.

```
#include <iostream>
#include <string>
using namespace std;

typedef enum E_TYPE { Er, Mgr, Dir } E_TYPE;
class Engineer {
protected: string name_, E_TYPE type_;
public: Engineer(const string& name, E_TYPE e = Er) : name_(name), type_(e) {}
    E_TYPE GetType() { return type_; }
    void ProcessSalary() { cout << name_ << ": Process Salary for Engineer" << endl; }
};

class Manager : public Engineer { Manager* reports_[10];
public: Manager(const string& name, E_TYPE e = Mgr) : Engineer(name, e) {}
    void ProcessSalary() { cout << name_ << ": Process Salary for Manager" << endl; }
};

class Director : public Manager { Director* reports_[10];
public: Director(const string& name) : Manager(name, Dir) {}
    void ProcessSalary() { cout << name_ << ": Process Salary for Director" << endl; }
};

int main() { Engineer e1("Rohit"), e2("Kavita"), e3("Shambhu");
    Manager m1("Kamala"), m2("Rajib"); Director d("Ranjana");
    Engineer *staff[] = { &e1, &e1, &e2, &e3, &d };
    for (int i = 0; i < sizeof(staff) / sizeof(Engineer*); ++i) {
        E_TYPE t = staff[i]->GetType();
        if (t == Er) staff[i]->ProcessSalary();
        else if (t == Mgr) ((Manager*)staff[i])->ProcessSalary();
        else if (t == Dir) ((Director*)staff[i])->ProcessSalary();
        else cout << "Invalid Staff Type" << endl;
    }
    return 0;
}
```

At the bottom of the slide, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das". There is also a small number "10" in the bottom right corner.

If we do that then in terms of the type I still need to add the director type and in terms of here I need to have a director class which is a specialization of the manager, it may have another field which gives the reporting managers and so on. And when a director is constructed then you actually put a name and invoke the manager constructor and pass this director type here. If the process salary is again overridden for the director and now you have the logic for processing director's salary. Rest of it remains same in the application we have instantiated a director we have added that the director to the array. But if you look into the actual application code here, then you see that earlier we had these two lines because we had only engineer and manager now you have a Director.

So if my type t fails to match Er and Mgr I have to then check for whether it matches the Dir. If it matches the Dir then again I will have to do something risky I have to take this staff pointer which is engineer pointer and forcibly cast it to a director pointer so that this whole thing now becomes a pointer to director. And I invoke the process salary which will invoke the process salary for the director.

You can see that like it was in the case of C it is possible to keep on extending adding newer and newer types of employees, but in terms of quiet a bit of cost, in terms of quiet a bit of vulnerability and possible error because I have to manage the type explicitly by myself I have to propagate that properly and I have to every time I add a type my application code has to change. So just think of that if I have tens and hundreds of different types coming in then it is how difficult and how cumbersome this is going to be. Let us see of whatever C++ we have learnt whether we can have a better design.

(Refer Slide Time: 14:34)

The slide title is "C++ Solution: Non-Polymorphic Hierarchy" and the subtitle is "Engineer + Manager + Director". The slide content shows code for initializing objects and their output. The code is:

```
Engineer e1("Rohit"), e2("Kavita"), e3("Bhanu");
Manager m1("Kamala"), m2("Rajib"); Director d("Ranjana");
Engineer *staff[] = { &e1, &m1, &m2, &e2, &m3, &d };
```

The output is:

```
Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Kavita: Process Salary for Engineer
Bhanu: Process Salary for Engineer
Ranjana: Process Salary for Director
```

The sidebar on the left lists course modules: Module 30, Partha Pratim Das, Objectives & Outline, Staff Salary Processing, C Solution, C++ Solution, Non-Polymeric Hierarchy, Polymorphic Hierarchy (OOPs), and Summary. The footer includes the NPTEL logo, "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and slide number 11.

This is output you can check later on

(Refer Slide Time: 14:38)

The slide title is "C++ Solution: Polymorphic Hierarchy" and the subtitle is "Engineer + Manager + Director". The slide content shows a class hierarchy diagram and a list of questions. The diagram shows three classes: Director, Manager, and Engineer, connected by arrows pointing from Director to Manager and Manager to Engineer. The list of questions is:

- How to represent Engineers, Managers, and Directors?
 - Polymorphic class hierarchy
- How to initialize objects?
 - Constructor / Destructor
- How to have a collection of mixed objects?
 - array of base class pointers
- How to model variations in salary processing algorithms?
 - Member functions
- How to invoke the correct algorithm for a correct employee type?
 - Virtual Functions

The sidebar on the left lists course modules: Module 30, Partha Pratim Das, Objectives & Outline, Staff Salary Processing, C Solution, C++ Solution, Non-Polymeric Hierarchy, Polymorphic Hierarchy (OOPs), and Summary. The footer includes the NPTEL logo, "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and slide number 12.

Next design works with a polymorphic hierarchy. So what you change, we are again with the director, manager, engineer hierarchy, but what you change is we change from non-polymorphic to polymorphic class hierarchy.

Rest of these remains same, this remains same, this remains same, but the movement we change to polymorphic class hierarchy then our dispatch mechanism can change to virtual functions. I do not need, because this is precisely what the virtual functions are for, that if I have a base class pointer and I am pointing to a certain derived class object then I can call a function on the derived class object blindly if that function is a virtual function. So this is the precise job that the functions switch was doing which we can now realize in terms of the virtual function let us see how.

(Refer Slide Time: 15:29)

```

Module 30
Partha Pratim Das
Objectives & Outcome
Staff Salary Processing
C Solution
C++ Solution
Ans
Polymorphic Hierarchy
Polymorphic Hierarchy (Module)
Summary

NPTEL MOOCs Programming in C++
Partha Pratim Das
13
#include <iostream>
#include <string>
using namespace std;

class Engineer { protected: string name_; public: Engineer(const string name) : name_(name) {} virtual void ProcessSalary() { cout << name_ << ": Process Salary for Engineer" << endl; } };
class Manager : public Engineer { Manager *reports_[10]; public: Manager(const string name) : Engineer(name) {} void ProcessSalary() { cout << name_ << ": Process Salary for Manager" << endl; } };
class Director : public Manager { Manager *reports_[10]; public: Director(const string name) : Manager(name) {} void ProcessSalary() { cout << name_ << ": Process Salary for Director" << endl; } };
int main() { Engineer e1("Rohit"), e2("Kerita"), e3("Bhavinder"); Manager m1("Manal"), m2("Rajib"); Director d1("Sanjana"); Engineer *staff[] = { &e1, &m1, &m2, &d1, &e2 }; for (int i = 0; i < sizeof(staff) / sizeof(Engineer*); ++i) staff[i]->ProcessSalary(); return 0; }

```

We have the designed again this is the enumeration of the implied type is gone so is the related field in each one of this classes we just have the name, we just have the constructor setting the name the processing salary. The manager is a specialization of employee, director is a specialization of manager and so on. So certainly all those type information is gone because now I do not need to maintain enumerated type values to know what kind of object I have. The class itself will maintain that value will contain that value for me.

In terms of creating the object and in terms of you know setting the store nothing changes, but look at the application code this follow (Refer Time: 16:19) same, but this application code has become just this much. How will this work? This will work because

of this dynamic binding and dispatch mechanism of virtual functions. So what will happen? If I am going through this when I am 0 when i is 0 I have &e1 which is an employee pointer. What I do have on one side I have staff [i], so this has a static type which is engineer star. All invocations are with that so it tells me that whenever I try to invoke I will always start looking in the base class.

So what is a function that is being invoked, which is processsalary(), which is a member function in the base class and that member function is virtually. What will happen? If I have i= 0 then the actual pointed object at staff [0] is e1 which is an employee object. Because this is a virtual function the call will get delegated to the type of the object pointed too, so this function will get called which is the correct one. Think about the next one when i is 1, when i is 1 it is actually a manager pointer. When I try to do this dispatch I again start here this function this virtual and I know that the actual object been pointed to is a manager object. This virtual function will get delegated because manager classes overriden this function with the logic of how to compute manager salary and this will correctly invoke the process salary of the manager.

So that is the basic beauty off. The switching that we needed to do explicitly by maintaining type and then doing and if else in the application code is now all assumed in the basic feature of virtual functions where basic feature of dynamic dispatch. So that is the power of dynamic dispatch that makes the designs really better. If I continue with two this will again be same this function will get called because it is a manager object 3, this will get called because in employee object 3, this will get called because is an employee object.

If we come here with d it is a director object. So again from staff [i] we will come to class engineer and then in class engineer again we will see this is a virtual function. So we will try to delegate it to the type of the object that is the class director. And the class director has overriden in this function. In case of this last pointer the invoke function would be the process salary in the director class which will have the logic for how to process directors salary.

We can see that in two steps first moving onto C++ which gives us a lot of benefit in terms of putting in encapsulation then having constructor destructors to take care of in initialization, d initialization, then being able to create an array of base class pointers to create a convenient store and so on. We got one set of advantages and then when we move in the C++ design from the non-polymorphic hierarchy to polymorphic hierarchy we reap the maximum benefit in the designing terms of being able to support a dynamic dispatch which is built in to the language which is not to be coded by the application.

So you can see from the structure of this code, that here no where all that you need to know that it is of type engineer which is a base type. But if you if you add three more specializations from engineer you do not need to make any changes in this application code it can remain exactly the same you do not even need to recompile that, because you just need to recompile this class parts and link with it, because the whole dispatch mechanism is in terms of the definition of the virtual functions as they are put in the class hierarchy. So, we have made quite some progress in terms of this.

(Refer Slide Time: 20:56)

C++ Solution: Polymorphic Hierarchy
Engineer + Manager + Director

```
Engineer *e1("Rohit"), *e2("Kavita"), *e3("Shambhu");
Manager *m1("Kamala"), *m2("Rajib"); Director *d("Ranjana");
Engineer *staff[] = { e1, e2, m1, m2, e3, d };

Output:
Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Kavita: Process Salary for Engineer
Shambhu: Process Salary for Engineer
Ranjana: Process Salary for Director
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 14

This is the output for you to check.

(Refer Slide Time: 20:59)

The slide title is "C++ Solution: Polymorphic Hierarchy (Flexible) Engineer + Manager + Director + Others". The diagram illustrates a polymorphic hierarchy with an abstract base class `Employee` and its subclasses `Engineer`, `Manager`, and `SalesExecutive`. `Director` is shown as another class that interacts with the hierarchy. A list of questions follows:

- How to represent Engineers, Managers, Directors, etc.?
 - Polymorphic class hierarchy with an Abstract Base Employee
- How to initialize objects?
 - Constructor / Destructor
- How to have a collection of mixed objects?
 - array of base class pointers
- How to model variations in salary processing algorithms?
 - Member functions
- How to invoke the correct algorithm for a correct employee type?
 - Virtual Functions (Pure in Employee)

Module 30
Partha Pratim Das
Objectives & Outline
Staff Salary Processing
C Solutions
C++ Solutions
Polymorphic Hierarchy
Polymorphic Hierarchy (Flexible)
Summary

NPTEL MOOCs Programming in C++ Partha Pratim Das 15

Now to the last part. So, what does the company want us to do? The last specialization was that in future they might want to add some division and in that division they might want to add more types of employees. So far our hierarchy was only this part. If they add another division possibly something like sales division possibly something like sales executive will come in. We do not know in future what all divisions they are want to add, we do not know what are the different types of employees they are going to add; we do not know how many specializations of these employee types will exist and so on.

So, it becomes imperative, but we have all just seen that if we can have all this classes on a singly routed polymorphic hierarchy then my whole representation my whole code becomes very convenient to write. So we introduce a new route which they have not said, which the specification has not said we say let them be a concept call employee. Now there is nobody called employee in that organization. They are called engineer, manager, director and possibly in feature they will be some sales executive whatever. But we introduce there is a concept called employee and we said that let this be an abstract concept. There will be nobody who is an employee, but anybody who will perform the rule of an employee will be a specialization of this concept, specialization of this class there is a basic notion of the abstract base class.

So, we have now saying that let us extend this polymorphic hierarchy with abstract base class. Further, if this is a concept alone if there is no physical employee; which match this class then certainly it will mean that the processing salary logic for this particular class will not be known. So the process salary function member function that we had been using that will not be known for this employees, and that very nicely fits in to the abstractness of the concepts and this is not known all that we will need to do is to make that process salary a pure virtual function in this class because we will have no obligation to provide the processing logic for the processing of the salary in this class. So that is the change in the design that we bring in for the further extension we were working with a polymorphic class hierarchy now we have a polymorphic class hierarchy with an abstract base class.

All of these remain same, but in terms of virtual functions certainly we will have an additional virtual function in the employee class the route class which is where the all dispatch will enter which will be made a pure virtual function. So that is the basic design considerations that go in and let us look at the solution.

(Refer Slide Time: 23:59)

```

Module 30
Partha Pratim Das
Objectives & Outline
Staff Salary Processing
C Solution
C++ Solution
More Polymorphic Hierarchy Examples
Polymorphic Hierarchy (Flexible)
Summary

#include <iostream>
#include <string>
using namespace std;

class Employee {protected: string name_;}
public: virtual void ProcessSalary() = 0;
};

class Engineer: public Employee {public: Engineer(const string& name) { name_ = name; }
void ProcessSalary() { cout << name_ << ": Process Salary for Engineer" << endl; }
};

class Manager: public Engineer {public: Manager(const string& name) : Engineer(name) {}
void ProcessSalary() { cout << name_ << ": Process Salary for Manager" << endl; }
};

class Director : public Manager { Manager* reports_[10];
public: Director(const string& name) : Manager(name) {}
void ProcessSalary() { cout << name_ << ": Process Salary for Director" << endl; }
};

class SalesExecutive : public Employee {public:
SalesExecutive(const string& name) { name_ = name; }
void ProcessSalary() { cout << name_ << ": Process Salary for Sales Executive" << endl; }
};

int main() {
Employee e1("Rohit"), e2("Kavita"), e3("Shambhu");
Manager m1("Komal"), m2("Bajibh"); SalesExecutive s1("Hari"), s2("Bishnu");
Director d("Ranjan");
Employee *staff[] = { &e1, &e1, &m2, &s1, &s3, &d, &m2 };
for (int i = 0; i < sizeof(staff) / sizeof(Employee*); ++i) staff[i]->ProcessSalary();
return 0;
}

```

NPTEL MOOCs Programming in C++ Partha Pratim Das 16

This part engineer to director there is no change except that we have introduced a new base class. Since we have introduced a new base class the engineer is now specialization

from that. Since we have done that we have moved the data member here. What is the need for moving the data member here, because earlier see if we just add an employee class and let us say we allow engineer to specialize for that and have the name here then certainly there will be no issue as you go to the manager and director and so on.

But once you open another division and here I have sells executive I will need to have name for the sales executive here as well. So, I will have duplication of code. How do you factor this out? You just factor this out by moving them to the common part. So that is why you get the name moving up to the abstract base class. So having said that then we have added another class sales executive which possibly comes from the new division which is yet not there, but we are just trying out our code and you see that the sales executive again directly inherits from employee because it is going on a different part of the hierarchy.

Whereas, manager continues to inherit from the engineer, director continues to inherit from manager and so on, and each one of them has different implementation for the processing logic. And what binds them together, is put the processing logic member function this virtual member function in the base class employee and make that pure.

So that ensures that no employee instance can be constructed which gives us comfort because we do not know what would it have meant to construct an instance of an employee because we do not have any details about that. But it does allow us that we can now create our collection of employees as in employee array of employee pointer, earlier we were doing with array of engineer point. Now it is an array of employee pointer this pointers so that that is something very very beautiful you are you cannot actually construct an object, but you can always have a pointer of that type because pointer does not need an instance you just skipping an address and thinking about that address.

So, staff is thought of as if of this type and will use that information for doing the polymorphic dispatch, but it will actually have objects where none of them is of employee take their all of the specialize type which are concrete classes.

So, that makes the whole thing buying together multiple stands buying together. Interesting there is no change in this follow there is no change in this application code from what we did with a polymorphic hierarchy which did not have an abstract base class in the route. So, there is no impact in terms of the application code there is no impact in terms of any of the earlier classes that we have, there is no impact in terms of the container in which we have putting the object. But we have been able to further reduce refactor the code make it shorter and you had made it possible that from the employee any kind of other hierarchies could be made to specialize and my application code, my basic information here will not need to change.

So that is the power of polymorphic hierarchy particularly when you need to extend the designs which we often need to do and particularly when we use it with certain route classes which are abstract so that we can combined concepts which may not always be concretely realizable in terms of abstract base classes, but still can define operations for them, still can define pointers to those abstract base classes and use those pointers and the actual instance at the runtime to a dynamically bound navigation and a polymorphic dispatch for calling the actual member function for the object instance that we have at hand.

(Refer Slide Time: 29:02)

**C++ Solution: Polymorphic Hierarchy (Flexible)
Engineer + Manager + Director + Others**

```
Module 30
Partha Pratim Das
```

Objectives & Outline

- Staff Salary Processing
- C Solution
- C++ Solutions
- Mac
- Polymorphic Class
- Polymorphic Hierarchy
- Summary

Output:

```
Engineer e1("Rohit"), e2("Kavita"), e3("Shambhu");
Manager m1("Rasika"), m2("Raajib"); SalesExecutive s1("Hari"), s2("Bishnu");
Director d("Banjana");
Employee *staff[] = { &e1, &m1, &s2, &e1, &m2, &d, &s2 };
```

```
Rohit: Process Salary for Engineer
Rasika: Process Salary for Manager
Raajib: Process Salary for Manager
Kavita: Process Salary for Engineer
Hari: Process Salary for Sales Executive
Shambhu: Process Salary for Engineer
Banjana: Process Salary for Director
Bishnu: Process Salary for Sales Executive
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 17

At present this is our final design, we might want to see if we want more refinements of that I would really like if you all you try to work this out and if you think that this design could be improved in some ways.

(Refer Slide Time: 29:11)

Then you can just write on the forum you can take up and discuss that. But for now this is the completion of our design exercise on the staff salary application problem. In this we have tried to show a complete landscape of possible designs and solutions that you could do starting from C and in C2 we started and showed how the whole thing and still be modeled, but there are several lacunae (Refer Time: 29:39).

Then you move to C++ and in this we have shown three stages of the solution - first, with non-polymorphic hierarchy; then, with polymorphic hierarchy; and then with a polymorphic hierarchy with abstract base class. And I hope that will give you lot of strength in terms of being able to do more designs in future.

In the next module and two we will take a brief look into all that we are saying in terms of the polymorphic dispatch the dynamic binding, how does that actually work, and then continue with other features of C++.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 46
Virtual Function Table

Welcome to Module 31 of Programming in C++. In the last couple of modules we have discussed about Dynamic Binding, the Polymorphism, we have discussed what is polymorphic type and particularly the major new feature that we have learnt is about polymorphic dispatch.

It is a mechanism through which, when I call a member function of a class belonging to a polymorphic hierarchy. When I make the call through a pointer or through the reference of the base class type then the actual function invoked depends not on the type of the pointer or the reference, but it actually depends on the current object being pointed to or being referred to. It depends on the runtime and this mechanism is known as Polymorphism or this method of binding is known as Dynamic Binding as we have seen, and this whole thing is known as Polymorphic Dispatch in C++.

(Refer Slide Time: 01:37)

The screenshot shows a presentation slide with a blue header bar. On the left, there is a sidebar with a logo at the top, followed by 'Module 31', 'Partha Pratim Das', and 'Objectives & Outline'. Under 'Objectives & Outline', there are several items listed: 'Staff Salary Processing', 'C Behavior', 'C++ Behavior', 'Virtual Function Pointer Table', and 'Summary'. The main content area has a title 'Module Objectives' and a bullet point: '• Understand Virtual Function Table for dynamic binding (polymorphic dispatch)'. At the bottom of the slide, there is footer text: 'NPTEL MOOCs Programming in C++', 'Partha Pratim Das', and the number '2'.

We have looked at polymorphic dispatch mechanism and we have used it extensively in designing a solution of staff salary processing. In the module today, we will discuss about Virtual Function Table that is we would like to take a little bit of understanding in terms of how is this polymeric dispatched; actually implemented by the compiler. Because you will have to recall that the compiler works at the static time when the code is being process; the source is being processed at that time there is no execution that is happening.

So, compiler has no way of knowing as to at runtime given a pointer which is the type of object that it will actually point to, but still it has to generate a code that will work according to the dynamic type during the execution.

(Refer Slide Time: 02:44)

The screenshot shows a presentation slide titled "Module Outline" for "Module 31". The slide features a sidebar with navigation links: "Module 31", "Partha Pratim Das", "Objectives & Outline", "Staff Salary Processing C Solution", "C++ Solutions", "Virtual Function Pointer Table", and "Summary". The main content area displays a bulleted list of topics:

- Staff Salary Processing: RECAP
 - C Solution using Function Pointers
 - C++ Solution using Polymorphic Hierarchy
 - Comparison of C and C++ Solutions
- Virtual Function Table for Polymorphic Dispatch

At the bottom of the slide, there is a video player interface showing a circular thumbnail of a person's face, likely the speaker, with the text "Partha Pratim Das" next to it. The video player also displays the text "NPTEL MOOCs Programming in C++".

This is done through the use of virtual function table, which is key for the polymeric dispatch. For this we will take a quick recap of the salary processing application we have been discussing. We will introduce a new solution in C and show how that gives us the insight into understanding the virtual function table for the dispatch.

(Refer Slide Time: 03:06)

The slide title is "Staff Salary Processing: Problem Statement: RECAP (Module 29)". The content lists requirements for a salary processing application:

- An organization needs to develop a salary processing application for its staff
- At present it has an engineering division only where Engineers and Managers work. Every Engineer reports to some Manager. Every Manager can also work like an Engineer
- The logic for processing salary for Engineers and Managers are different as they have different salary heads
- In future, it may add Directors to the team. Then every Manager will report to some Director. Every Director could also work like a Manager
- The logic for processing salary for Directors will also be distinct
- Further, in future it may open other divisions, like Sales and expand the workforce
- Make a suitable extensible design**

NPTEL MOOCs Programming in C++ Partha Pratim Das

So this was the problem, we had assumed that there are different types of employees for whom a salary needs to be processed and their salary processing algorithms are different. And the crux of the whole design is the fact that the design needs to be appropriately extensible. It should be possible to extend and add new classes to the hierarchy as and when we want to do that.

(Refer Slide Time: 03:32)

The slide title is "C Solution: Function Pointers Engineer + Manager: RECAP (Module 29)". The content lists how to represent Engineers, Managers, and Directors:

- How to represent Engineers, Managers, and Directors?
 - struct
- How to initialize objects?
 - Initialization functions
- How to have a collection of mixed objects?
 - Array of union
- How to model variations in salary processing algorithms?
 - struct-specific functions
- How to invoke the correct algorithm for a correct employee type?
 - Function switch
 - Function pointers

NPTEL MOOCs Programming in C++ Partha Pratim Das

In C, we had taken this considerations and you can always refer to the earlier module, refer them have a module number also and check up on what are design considerations.

(Refer Slide Time: 03:48)

```
#include <stdio.h>
#include <string.h>

typedef union E_TYPE { Eng, Mgr, Dir } E_TYPE;
typedef void (*pFuncPtr)(void *);

typedef struct Engineer { char *name,_; } Engineer;
Engineer *InitEngineer(const char *name) { Engineer *e = (Engineer *)malloc(sizeof(Engineer));
e->name_ = strdup(name); return e;
}
void ProcessSalaryEngineer(void *v) { Engineer *e = (Engineer *)v;
printf("%s: Process Salary for Engineer\n", e->name_);
}

typedef struct Manager { char *name,_; Engineer *reports_,[10]; } Manager;
Manager *InitManager(const char *name) { Manager *m = (Manager *)malloc(sizeof(Manager));
m->name_ = strdup(name); return m;
}
void ProcessSalaryManager(void *v) { Manager *m = (Manager *)v;
printf("%s: Process Salary for Manager\n", m->name_);
}

typedef struct Director { char *name,_; Manager *reports_,[10]; } Director;
Director *InitDirector(const char *name) { Director *d = (Director *)malloc(sizeof(Director));
d->name_ = strdup(name); return d;
}
void ProcessSalaryDirector(void *v) { Director *d = (Director *)v;
printf("%s: Process Salary for Director\n", d->name_);
}
```

Now, while we discussed the solution we had used the mechanism of a function switch, that is we explicitly maintained the type and at the time of processing the record of every employee we checked on what is the type maintaining that in a union and then based on the type we call the appropriate salary processing function. Now we would try to look at the same solution with a little bit of different flavor using function pointers. The main differences that, I would highlight; certainly the explicit maintenance of types stills remains.

I am just assuming that there are three types; engineer, manager, and director. Earlier if you recall you may open up the earlier video or presentation to check that the salary processing routines, say the salary processing for the engineer was taking a pointer to engineer. Salary processing for manager was taking a pointer to manager, and salary processing to the director was taking a pointer to director. Now I have changed it a little bit all of them now take a void star pointer, because I want all of these functions to have the same signature.

Once we get it as void star naturally, the void star does not tell me what type of object it is, but since I know the salary processing for engineer has been invoked I know that void star pointer actually points to an engineer record, so I cast it to the engineer. Or if it is a manager function called I cast it to manager or director function called I cast it to director and then do the processing. I can still manage without actually explicitly having different parameter types here. But in this process what a gain is; all this functions now have the same signature.

Since they have the same signature I can try to combine them in terms of a uniform function pointer type which I say is a ptr is a function pointer which takes a void star and does not written anything. Note this is a function pointer so this basically is a function pointer type. this is not a function itself. We will define function.

So, any of this functions for engineer for manager or for the director will actually match with this function pointer type. Rest of the design remain same, the different structure types to define different types of objects and with this let us see how do we actually combine the whole thing into the application.

(Refer Slide Time: 06:33)

```

Module 31
Partha Pratim Das
Objectives &
Outline
Staff Salary
Processing
C Solution
C++ Solution
Virtual
Functions
Dynamic Table
Summary

C Solution: Function Pointers
Engineer + Manager + Director

typedef struct Staff {
    E_TYPE type;
    void *p;
} Staff;

int main() {
    void (*ptrarr[5])() = { ProcessSalaryEngineer,
                           ProcessSalaryManager,
                           ProcessSalaryDirector };

    Staff staff[] = { { Er, InitEngineer("Rohit") },
                      { Mgr, InitEngineer("Kamala") },
                      { Mgr, InitEngineer("Rajib") },
                      { Er, InitEngineer("Kavita") },
                      { Er, InitEngineer("Bhambhani") },
                      { Dir, InitEngineer("Ranjanan") } };

    for (int i = 0; i < sizeof(staff) / sizeof(Staff); ++i)
        ptrarr[staff[i].type](staff[i].p);

    return 0;
}

Output:
Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Kavita: Process Salary for Engineer
Bhambhani: Process Salary for Engineer
Ranjanan: Process Salary for Director

```

NPTEL MOOCs Programming in C++ Partha Pratim Das

In terms of the application what we do is, we now maintain a record where on one, it say, it is basically a doublet this is where I keep the type which would be something like Er or Mgr and so on. The other is a pointer of void star that is this is a pointer of some kind. Now what we do? We populate an array of function pointer. So this is an array of function pointer. I have already defined a function pointer type. So this is an array of function pointers 0, 1 and 2, and in each one of these we actually assign three different functions that we have written for the three different structure types.

These have three different function pointers actually deciding in the array. Then you populate the collection of objects, so collection of objects is the object type and the object instance. It is a pointer to the object instance so in place of void star this will now have. If I look into this array this is like doublet, so I have Er then an Er object, I have Mgr, I have a manager object and so on. So we will have in this way, there are six different of them. This stuff is my total collection.

Now, when I want to process that I will go over a loop and this is what is critical is I will use this function pointer array and using the type of i-th staff. Say if i is 0 when the type is Er. I will pick up from this function pointer array that is this array, I will index it by this type which is Er that means 0 and then we will call that function. So, this psArray if I just use a different color, say this particular one is a function pointer, because psArray is this array. I have indexed it with the stuff i dot type which is Er in this case, the first one is an engineer. So what I get is a function pointer which is a function of this is the type of function, it takes a void star and the function is called. And what it passes is the second parameter here which is staff [i].p; the pointer to the object.

When I call it with Er here, so I have got this function that is process salary engineer this will get called by the engineer record, that is this will get called with the in it engineer that have done with the Rohit. When i becomes 1, I get the second record where I have the type as Mgr and which is 1, so I get the second function pointer and I invoke it with the pointer to the corresponding employee which is the manager record. In this way one after the other all of this functions will get called with the appropriate type with the appropriate employee record and once we are inside the record we have already seen that

if the engineer function has been invoked and I have got a to the pointer to the engineer record here then I will cast and execute that.

This is the basic style of doing this. The change that we have made is; we have made this whole thing into array of function pointers here and we are invoking them through a simple code. And we do not need any more the switch; conditional switch that we had been doing all that is taken care of by the indexing in this function pointer array. This could be another solution in C which is a smart solution in C which will be very useful.

(Refer Slide Time: 11:26)

The screenshot shows a presentation slide with the following details:

- Title:** C++ Solution: Polymorphic Hierarchy: RECAP
Engineer + Manager + Director: (Module 30)
- Navigation:** Module 31, Partha Pratim Das, Directions & Outline, Staff Salary Processing C Solution, C++ Solution, Virtual Function Pointer Table, Summary.
- Diagram:** A class hierarchy diagram showing Director, Manager, and Engineer classes connected by arrows.
- Topics:**
 - How to represent Engineers, Managers, and Directors?
 - Polymorphic class hierarchy
 - How to initialize objects?
 - Constructor / Destructor
 - How to have a collection of mixed objects?
 - array of base class pointers
 - How to model variations in salary processing algorithms?
 - Member functions
 - How to invoke the correct algorithm for a correct employee type?
 - Virtual Functions
- Footer:** NPTEL MOOCs Programming in C++, Partha Pratim Das, and a portrait photo of Partha Pratim Das.

If we just quickly compare it with the polymeric C++ solution this was basic approach to the solution.

(Refer Slide Time: 11:36)

The slide title is "C++ Solution: Polymorphic Hierarchy: RECAP Engineer + Manager + Director: (Module 30)". The sidebar on the left lists: Module 31, Partha Pratim Das, Objectives & Outline, Staff Salary Processing, C Solution, C++ Solution, Virtual Function, Pointer Table, Summary. The main content area contains the following C++ code:

```
#include <iostream>
#include <string>
using namespace std;

class Engineer {
protected:
    string name_;
public:
    Engineer(const string& name) : name_(name) {}
    virtual void ProcessSalary() {
        cout << name_ << ": Process Salary for Engineer" << endl;
    }
};

class Manager : public Engineer {
    Engineer *reports_[10];
public:
    Manager(const string& name) : Engineer(name) {}
    void ProcessSalary() {
        cout << name_ << ": Process Salary for Manager" << endl;
    }
};

class Director : public Manager {
    Manager *reports_[10];
public:
    Director(const string& name) : Manager(name) {}
    void ProcessSalary() {
        cout << name_ << ": Process Salary for Director" << endl;
    }
};
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

If you look at then you will see that we have all the same things the destructor, the constructor, the different overridden process salary functions and so on.

(Refer Slide Time: 11:53)

The slide title is "C++ Solution: Polymorphic Hierarchy: RECAP Engineer + Manager + Director: (Module 30)". The sidebar on the left lists: Module 31, Partha Pratim Das, Objectives & Outline, Staff Salary Processing, C Solution, C++ Solution, Virtual Function, Pointer Table, Summary. The main content area contains the following C++ code:

```
int main() {
    Engineer e1("Bohit");
    Engineer e2("Kavita");
    Engineer e3("Shaabhu");
    Manager m1("Kamala");
    Manager m2("Rajib");
    Director d1("Ranjana");

    Engineer *staff[] = { &e1, &m1, &m2, &e3, &d1 };
    for (int i = 0; i < sizeof(staff) / sizeof(Engineer); ++i) staff[i]->ProcessSalary();
    return 0;
}
```

Output:
Bohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Kavita: Process Salary for Engineer
Shaabhu: Process Salary for Engineer
Ranjana: Process Salary for Director

NPTEL MOOCs Programming in C++ Partha Pratim Das

And with that if we look at our code is simply for loop with this call being made. Where this each and every one of this is a pointer to the engineer type and depending upon the polymeric dispatch it will call the appropriate processSalary function.

(Refer Slide Time: 12:21)

C Solution	C++ Solution
<ul style="list-style-type: none"> How to represent Engineers, Managers, and Directors? structs 	<ul style="list-style-type: none"> How to represent Engineers, Managers, and Directors? Polymorphic hierarchy
<ul style="list-style-type: none"> How to initialize objects? Initialization functions 	<ul style="list-style-type: none"> How to initialize objects? Ctor / Dtor
<ul style="list-style-type: none"> How to have a collection of mixed objects? array of union wrappers 	<ul style="list-style-type: none"> How to have a collection of mixed objects? array of base class pointers
<ul style="list-style-type: none"> How to model variations in salary processing algorithms? functions for structs 	<ul style="list-style-type: none"> How to model variations in salary processing algorithms? class member functions
<ul style="list-style-type: none"> How to invoke the correct algorithm for a correct employee type? Function switch Function pointers 	<ul style="list-style-type: none"> How to invoke the correct algorithm for a correct employee type? Virtual Function

NPTEL MOOCs Programming in C++

Partha Pratim Das

If we compare this to side by side this is just comparing the design features.

(Refer Slide Time: 12:26)

C Solution (Function Pointer)	C++ Solution (Virtual Function)
<pre>#include <string.h> #include <assert.h> typedef enum E_TYPE { Er, Mgr, Dir } E_TYPE; typedef void (*pfnFunc)(void *);</pre>	<pre>#include <iostream> #include <string> using namespace std;</pre>
<pre>typedef struct { char *name; } Engineer; Engineer *InitEngineer(const char *name); void ProcessSalaryEngineer(void *v); typedef struct { char *name; } Manager; Manager *InitManager(const char *name); void ProcessSalaryManager(void *v); typedef struct { char *name; } Director; Director *InitDirector(const char *name); void ProcessSalaryDirector(void *v); int main() { pfnFunc pfnFunc[] = { ProcessSalaryEngineer, ProcessSalaryManager, ProcessSalaryDirector }; for (int i = 0; i < sizeof(staff)/sizeof(*staff); ++i) pfnArray[staff[i].type](&staff[i].p); return 0; }</pre>	<pre>class Engineer { protected: string name; }; public: Engineer(const string name); virtual void ProcessSalary(); }; class Manager : public Engineer { public: Manager(const string name); void ProcessSalary(); }; class Director : public Manager { public: Director(const string name); void ProcessSalary(); }; int main() { Engineer e1("Rohit"); Manager m1("Kousal"); Director d1("Ranjana"); Engineer *staff[] = { &e1, &m1, &d1 }; for(int i = 0; i < sizeof(staff)/sizeof(*staff); staff[i]->ProcessSalary(); } return 0; }</pre>

NPTEL MOOCs Programming in C++

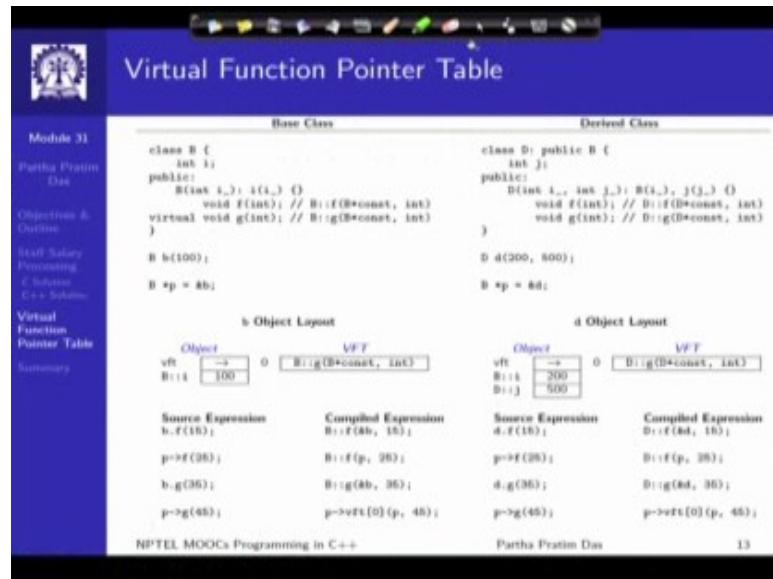
Partha Pratim Das

But if we just compare this to solution side by side you will see that this is the C solution this is the corresponding C++ class. So, you have the in it engineer, you have the constructor here, you have this particular processing function, you have the over load member function which is a polymorphic in nature. Similarly, I have for the next type the director type and these are three classes that we have.

Now, if we look at the processing part, see how similar this two processing look like. So here I take the pointer to that object and based on the dynamic type I dispatch to any one of this functions depending on its corresponding type. Here we do the same thing using the array psArray which is the array of the function pointers that can be used. The only difference is I need to explicitly maintain the type and the relationship between the type and the particular object that I am using.

So based on the type I pick up the function and then for that particularly the second part tells me what is the object pointer and we pass the object pointer. So, here I do not need to do that because as we know all that we need is to basically pick up the right type, the object pointer implicit as this pointer and that will work. The reason I show you this example or show you this way doing it in using function pointer in C is a fact that this is how actually in reality the C++ compiler takes care of the virtual functions or the polymorphic function.

(Refer Slide Time: 14:22)



So to bring it to the actual details, suppose I have a base class B, so for this base class I have two functions; one is function f and this function g. Which are; one is of polymorphic type g is polymorphic type, f is a non polymorphic type, it is a non-virtual function. And based on the this base class I have a derived class here which over writes the non-polymorphic function as well as the polymorphic function.

Now, what you do in this case is; For a non-polymorphic function let us say if we just talk about the function f which is non-polymorphic in nature, so how will the invocations look like? Suppose if I do sample b. f() which is directly accessing it with object. So, how will the compiler call it? Compiler will knows from the fact that b is a class B type of object the compiler knows that there is a function f so that is been called. So, it calls b colon-colon f which is this function, and what is required? It requires this pointer which is the address of b, so it puts the address of b and it puts the parameter. So, it puts the address of b here and the parameter and this will get called. This is a static binding; this is a simple static binding.

If I do that using a pointer I get the same thing doing it with a pointer. So, if I am calling this then from the type of the pointer which is of b type it knows that it has to look in the b class we have seen this before and in the b class it has f function which is non-virtual,

so it will directly have to call that function. So, it is statically puts `b::f()` passes the pointer value and the parameter. These are the static ways this is absolutely fine.

Now, let us look at a third invocation also where I am using the object and calling the non-virtual function `g`. Again the compiler does the same thing. We know that if we call it with the object it is the function of that class which will get called, so `b :: g()` is what we called. Address of `b` is passed here and the parameter is passed here. So, up to this point for these three all of these are basically static binding. So, if I instead of `b` if I have a derive class object `d` and I try to do this for the derive class object I will have similar static binding code here, where in each one of these the compiler knows that the type is `d` and whether it is directly by the object or it is through a pointer, but the function is a non virtual one so it will put the static calls.

The issue arises when I want to actually look at calling something like `p` pointer `g`, that is I am calling virtual function and I am using a pointer, so this is where I need the dynamic binding. I am calling `p` pointer `g`, I am using a pointer and I am using a virtual function. In this case the `p` is a pointer to a `b` object, in the case on right hand side `b` is pointer to the `d` object. So, what I will need here? I will need here that `p` this call will actually resolve to this function.

Whereas I will need that in this case this call will resolve to this function. Even though both of this calls at the call site looks same. How do I do that? how do I make that change? So what I do is something very very simple. Think about the layout of the object, the object has a data member `i`, so I have a data member `i` this was constructed with hundred. So, I have a data member `I`, `b :: i` which is the field here.

Similarly, if I look at the derive class object, it has added a data member `j` so it will have one base class part which is 200 and the additional data member that has added which is 500. So these are fine. Now what we do is we add one more field, this is invisible field we add one more filed to the object.

This field is a pointer filed and what does it point to, it points to a table of function pointers and this table is known as the VFT or the Virtual Function Table. So what it does

is, when say, I am in class b and I am trying to look at the codes, I have a virtual function pointer and I have one virtual function. So I put that virtual function in this table that is I put this pointer in this table. And whenever I will get a call for this function g this virtual function g through a pointer, I will not generate a code like this, I will not generate this static code rather I will generate a code like this. Do not get confused with this syntax, what is being said? P is a pointer. So p is a pointer, so p points here.

What is p pointer VFT? P pointer VFT is a pointed to this table. And if p pointer VFT is pointed to this table p pointer VFT 0 is a 0th entry of this table, and what is that entry? That entry is a function pointer. So, p pointer VFT 0 is this 0th function pointer in the table. So, I say that whenever you get p pointer g you actually call whatever function exists in this location. So you pick up this function and call it with the parameters, with parameters up to with this pointer which is p and the actual parameter value which is 45.

The compiler knowing that this is a virtual function and knowing that this is been called through a pointer does not generate the static time code like, hard codes the function, but it puts the call as if through the virtual function table. How does that help? The way it helps is now you have one level of indirection. So what happens is, when you have specialize this class b into class d and you construct a object for that that object also has a virtual function table pointer. That point to virtual function table, this table is now of class d, this table was of class b. Now what I have done in class d? In the class d this function virtual function g has been overridden a new definition has been given so that is now become d :: g.

So, while we specialize following the specialization we remove this function from the virtual function table of b and through the overriding we put the function that we have written new that is a function for d, d colon-colon g in the virtual function table.

And for this call, the compiler generates the same indirected code. Now what will happen? Now think off. This was the call and this is what the function is generated the compiler has generated these are call what has being generated this is what is been generating the static time. Now what happens, the two scenarios are that p is actually pointing to a b object. If it points to a b object it has the virtual function table pointer it

goes to this virtual function table of b this is what it has got, it picks of function 0 the 0th function and calls that, so it is calling `b :: g`.

But if p is pointing to a d object here then it has a different function pointer, virtual function table pointer. When it traverses at that pointer v p pointer VFT it actually gets the virtual function table of d. It picks up the 0th function which is this function, which happens to be now `d :: g` because it actually points to a d object so this table is different. Then it passes the parameters and call that naturally `d :: g` will get a called.

With this, simple mechanism of table of function pointer, so what we will learn from this, is if a class is polymorphic if a type is polymorphic that if it has at least one a virtual function then for that class there will be a virtual function table, which will have all the virtual functions listed one after the other in the table in the order in which they have been defined. Here we have only one entry because we have only one virtual function. If there are more then there will be more entries and as the class is specialized the compiler checks does the class redefine the virtual function if it does then it changes the corresponding entry in its own virtual function table and replaces with the version that has override.

And then with always makes a call instead of making a direct hard coded call like this a static type call like this where it clearly says what the function, it says that I do not know the about the function at the runtime go to the virtual function table pick up the 0th function and whatever function pointer is your function.

So, depending on the type of the object the virtual function table will be different, but by the same mechanism the current entry the runtime entry in the 0th location for this function will appropriately a point to either the function in the base class or the function in the derive class depending on which kind of object I have and therefore which kind of virtual function table I am pointing to. The basic virtual function I am pointer table mechanism by which we can do things.

(Refer Slide Time: 26:00)

The slide has a blue header bar with the title 'Virtual Function Pointer Table'. Below the header is a sidebar with navigation links: Module 31, Partha Pratim Das, Objectives & Outline, Staff Salary Processing, C Solutions, C++ Solutions, Virtual Function Pointer Table, and Summary. The main content area contains a bulleted list of points about VFT:

- Whenever a class defines a virtual function a hidden member variable is added to the class which points to an array of pointers to (virtual) functions called the **Virtual Function Table (VFT)**
- VFT pointers are used at run-time to invoke the appropriate function implementations, because at compile time it may not yet be known if the base function is to be called or a derived one implemented by a class that inherits from the base class
- VFT is class-specific – all instances of the class has the same VFT
- VFT carries the **Run-Time Type Information (RTTI)** of objects

At the bottom of the slide are footer links: NPTEL MOOCs Programming in C++, Partha Pratim Das, and 14.

As I said whenever a class defines a virtual function a hidden member variable is added to the class which points here. And at the runtime the invocation is indirectly through this and this carries what is known as RTTI. We will talk about RTTI more that is Run-Time Type Information of the whole polymorphic system.

(Refer Slide Time: 26:23)

The slide has a blue header bar with the title 'Virtual Function Pointer Table'. Below the header is a sidebar with navigation links: Module 31, Partha Pratim Das, Objectives & Outline, Staff Salary Processing, C Solutions, C++ Solutions, Virtual Function Pointer Table, and Summary. The main content area shows three diagrams labeled a, b, and c illustrating object layouts and VFT pointers:

a Object Layout

Object	vft	→	0	VFT
			0	A::f(A*const, int)
			1	A::g(A*const, double)

b Object Layout

Object	vft	→	0	VFT
			0	B::f(B*const, int)
			1	A::g(A*const, double)
			2	B::h(B*const, B*)

c Object Layout

Object	vft	→	0	VFT
			0	B::f(B*const, int)
			1	C::g(C*const, double)
			2	C::h(C*const, B*)

Source Expression **Compiled Expression**

pA->f(2);	pA->vft[0](pA, 2);
pA->g(3,2);	pA->vft[1](pA, 3,2);
pA->h(ka);	A::h(pA, ka);
pA->h(kb);	A::h(pA, kb);

pB->f(2);	pB->vft[0](pB, 2);
pB->g(3,2);	pB->vft[1](pB, 3,2);
pB->h(ka);	pB->vft[2](pB, ka);
pB->h(kb);	pB->vft[2](pB, kb);

At the bottom of the slide are footer links: NPTEL MOOCs Programming in C++, Partha Pratim Das, and 15.

This is one more example, the example that we had solved as a binding exercise so you can see here this is just for your further illustration. This is primarily for your own working out, but we have a class A and specialized from that we have class B, specialized from that we have class C. And class A has a polymorphic function it has a virtual function so naturally the whole hierarchy is polymorphic and therefore, the all objects on any of this classes on this hierarchy will have a virtual function table pointer.

So if I have the object A, say object A here then it will have data members those are not interesting so we have not listed those, but it will have a virtual function table pointer which will point to VFT of class A. And how many entries will be there? There is one virtual function f and one virtual function g. So there is a virtual function f there is a virtual function g.

But when it specializes to B you have made h also a virtual function. So what will happen? A third function will get added in the location two. And what will happen to 0 and 1? F was defined in A and there is over head name B. So, the 0th entry that was for f gets over hidden. Now, in place of a colon-colon f you have b :: f. G was also defined as a virtual function in a it resides in location one, but class b has not overridden g. So, when you inherit you actually get the same a :: g as function number one. And h you have made it a virtual function a fresh so the virtual function table gets added with an additional function pointer.

As you come to C object what you have done you have overridden g. So, what you get, you have not done anything with f, so f is simply inherited. So you got b :: f in the 0th entry continuous to be b colon-colon f. But this entry number one was a :: g, but now you have over written that, so that gets over written with c colon-colon g.

Entry number two was b ::. The h function in class b that have been over written here so you get c :: h, this is how the virtual function table will keep on growing and that basically will tell you why we said that once of function becomes virtual on this hierarchy it will have to remain virtual because once it that the compiler as a decision to make and here I have shown what will be the compile version of this codes are.

So you can say this goes to virtual function table entry 0, g goes to table entry 1, but where as if I do p a pointer h then I have static binding, the explicit function calls because this function was non-virtual. Whereas when I do it with p b, that is a pointed to the class b type then they have brought out it through the virtual function table because here in class b h has become a virtual function.

So please work this out, please try to understand this construction very carefully and work through this source expression and against the compile expression of where you have static binding and where you have dynamic binding so that you will be able to understand this as well.

(Refer Slide Time: 30:13)

The slide is titled "Module Summary". On the left, there is a sidebar with the following navigation links:

- Module 31
- Partha Pratim Das
- Objectives & Outline
- Staff Salary Processing
- C Solutions
- C++ Solutions
- Virtual Function Pointer Table
- Summary

Below the sidebar, the main content area contains two bullet points:

- Leveraging an innovative solution to the Salary Processing Application in C using function pointers, we compare C and C++ solutions to the problem
- The new C solution is used to explain the mechanism for dynamic binding (polymorphic dispatch) based on virtual function tables

At the bottom of the slide, there is a small circular portrait of the instructor, Partha Pratim Das, and the text "NPTEL MOOCs Programming in C4-I" and "Partha Pratim Das".

To summarize it leveraging and innovative solution to the staff salary application which uses function pointers in C we have laid the foundation for explaining how virtual functions are implemented using the virtual function pointer table. Please try to understand this more clearly so that any confusion about dynamic dispatch would be clear in your mind.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 47
Type Casting and Cast Operators: Part I

Welcome to Module 32 of Programming in C++. In this module and the next couple of ones, we will discuss in depth about type casting and cast operators in C++.

(Refer Slide Time: 00:41)

The screenshot shows a presentation slide with a dark blue header and footer. The header contains the Indian Institute of Technology Kharagpur logo and the title 'Module Objectives'. The footer includes the text 'NPTEL MOOCs Programming in C++', 'Partha Pratim Das', and the number '2'. The main content area is white and lists the objective: 'Understand casting in C and C++'.

Module Objectives

- Understand casting in C and C++

NPTEL MOOCs Programming in C++ Partha Pratim Das 2

So, the objective is to understand casting in C and C++.

(Refer Slide Time: 00:48)

The screenshot shows a presentation slide titled "Module Outline". On the left, there's a sidebar with the title "Module 32" and the name "Partha Pratim Das". Below this, under "Objectives & Outline", are several sections: "Casting", "Upcast & Downcast", "Cast Operators", "const_cast", and "Summary". The main content area contains a bulleted list of topics:

- Casting: C-Style: RECAP
 - Upcast & Downcast
- Cast Operators in C++
 - const_cast Operator
 - static_cast Operator
 - reinterpret_cast Operator
 - dynamic_cast Operator
- typeid Operator

At the bottom of the slide, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das". To the right of the slide, there's a small circular video player showing a person with glasses and a plaid shirt.

This will be the overall outline. We have already made some small introduction to casting while we were talking about polymorphism, primarily the C kind of casting. And the fact that on a hierarchy, up cast that is casting from a specialized class object to a generalized class type is safe; and downcast is not safe, it can lead to really serious errors we observed all of that.

So, we will take a quick recap on that. And then we will introduce what is known as cast operator in C++, there are four different operators we will discuss that. Ending with the dynamic casting scenarios, where casting is done based on the runtime type; and for that we will discuss another additional type id operator. So, for these modules, we will carry the same outline, thus parts to be covered in the specific module will be shown in blue as we have done here. And for that module the blue part will be visible on the left of your screen in every slide.

(Refer Slide Time: 02:16)

The screenshot shows a presentation slide titled "Type Casting". The slide has a blue header with the title and a sidebar on the left containing navigation links for the module. The main content area lists reasons for casting, types of conversions, and provides code examples. A handwritten note "T1 : T2" is written above the code snippet. A video player interface is visible at the bottom right.

Module 32
Partha Pratim Das
Objectives & Outline
Casting
Upcast & Downcast
Cast Operators
cast.cast
Summary

Type Casting

- Why casting?
 - Casts are used to convert the type of an object, expression, function argument, or return value to that of another type
- (Silent) Implicit conversions
 - The standard C++ conversions and user-defined conversions
- Explicit conversions
 - Type is needed for an expression that cannot be obtained through an implicit conversion more than one standard conversion creates an ambiguous situation
- To perform a type cast, the compiler
 - Allocates temporary storage
 - Initializes temporary with value being cast

```
double f (int i,int j) { return (double) i / j; }

// compiler generates:
double f (int i, int j) {
    double temp_i = i, temp_j = j; // Conversion
    return temp_i / temp_j;
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, let us step back and ask for type casting, why should we cast. So, casting is primarily required to convert the type of an object, type of an expression, a function, function argument, a return value etcetera to another type. So, the basic concept of C++ is strongly rooted int type that is any variable, any constant, any object that we use has a well defined type or is expected to have a well-defined type. So, as we involve them in different expressions in different computations, there is a need that, for computing an expression, I need an object of a certain type, but at hand I have an object of a different type, and that is where I need to cast.

As we have seen in the context of C that a large number of type casting is done through implicit conversion or what can be said as silent conversion, these are standard C++ conversion; standard in the sense that conversions that are between known types and therefore, have already been enumerated by the language designer. So, there is a rule which dictates as to how I can implicitly convert an integer to a double or a double to an integer, whether I can convert a pointer to a certain type into a pointer to void and so on.

And it will also have an implicit conversion, which is add on over C of user-defined conversions see you would recall that C did not exactly have user-defined type. So, the user-defined conversion was not a part of C, and that will come in here, and those can be

used as implicit conversions also.

The word implicit here particularly mean that when we do this kind of a conversion, we actually do not write anything extra in the source could to say that I am doing this kind of a conversion. But from the context of the expression, and the context of the object being used, you figure out that a conversion has happened. Naturally, in contrast, there is a whole lot of conversion, which is known as explicit conversion. Now of course, if we have the convenience of implicit conversion then why do we have explicit conversion? Explicit conversion is one where I explicitly write that I am making a conversion. So, if I x is of certain type T1, I would try to write out that I would like to take x to another type T2 convert it to another type T2, and write it out explicitly in the source.

When I do that then I say I am doing explicit conversion and the reason I do explicit conversion could be many for example, it could be that implicit conversion is making a conversion which is not what I want. It is converting it to it into something which I am not comfortable with or it could be that implicit conversion finds that given an object or given an expression there are multiple possible conversions and it becomes ambiguous to which one should be taken.

And in those cases, and which are quite a few, you may need to make use of the explicit conversion. It is always a good practice to make conversions as explicit as possible, so that your intention in writing the code and what the compiler has understood for your intention becomes completely resolved, if you making an explicit conversion. Because you are writing it out in the part of the text if it is implicit then it depends on your understanding and the compilers understanding of that particular situation of conversion, and we will see how that may lead to different difficulties.

(Refer Slide Time: 06:50)

The screenshot shows a presentation slide titled "Type Casting". The slide content includes:

- Module 32
- Partha Pratim Das
- Objectives & Outline
- Casting
- Upcast & Downcast
- Cast Operators
- cast, cast
- Summary

Type Casting

- Why casting?
 - Casts are used to convert the type of an object, expression, function argument, or return value to that of another type
- (Silent) Implicit conversions
 - The standard C++ conversions and user-defined conversions
- Explicit conversions
 - Type is needed for an expression that cannot be obtained through an implicit conversion more than one standard conversion creates an ambiguous situation
- To perform a type cast, the compiler
 - Allocates temporary storage
 - Initializes temporary with value being cast

double f (int i,int j) { return (double) i / j; }

// compiler generates:
double f (int i, int j) {
 double temp_i = i, temp_j = j; // Conversion
 return temp_i / temp_j;
}

NPTEL MOOCs Programming in C++ Partha Pratim Das

A small video window in the bottom right corner shows a man speaking.

Now, to perform a conversion, to perform a type cast a compiler so far what the kind of things that you have seen, I have just try to illustrate one here that suppose we have written a function like this even in C. So, it is `f` is a function that takes two integers `int i` and `int j` and returns a double value and how does it return a double value by doing this. So, what all you have here is this is a nice illustration of the different conversions this is one where I make an explicit conversion. I say that `i` is an `int`, so `i` is of type `int`, I want to take it to `double`.

Now what happens to `j`? `j` is also of type `int`. Now I do not say that actually I also need `j` to be taken to the type `double`, because I am doing a division where the first argument, the first operand is a `double`. And therefore, the second operand also needs to be `double`. So, there will be another conversion here which is implicit. So, this is a simple example where you have implicit as well as explicit conversion.

(Refer Slide Time: 08:21)

The slide is titled "Type Casting" and is part of "Module 32" by Partha Pratim Das. The navigation menu on the left includes "Objectives & Outline", "Casting", "Upcast & Downcast", "Cast Operators", "cast.cast", and "Summary". The main content discusses casting, with sections on "Why casting?", "(Silent) Implicit conversions", and "Explicit conversions". It explains that type conversion creates ambiguous situations and shows code examples with handwritten annotations. The code example is:

```
double f (int i, int j) { return (double) i / j; }

// compiler generates:
double f (int i, int j) {
    double temp_i = i, temp_j = j; // Conversion in temporary
    return temp_i / temp_j;
}
```

Annotations include:
i = 2
j = 3
2.0
3.0

And as I have seen with a large number of my students, in students mind the thinking always is that i is an integer possibly its value was 2; j is another integer possibly its value was 3. And when I just convert it to double then I say that it becomes 2.0 or it will become 3.0 and so on and nothing actually happens, but that is not the case.

(Refer Slide Time: 08:40)

The slide is titled "Type Casting" and is part of "Module 32" by Partha Pratim Das. The navigation menu on the left includes "Objectives & Outline", "Casting", "Upcast & Downcast", "Cast Operators", "cast.cast", and "Summary". The main content discusses casting, with sections on "Why casting?", "(Silent) Implicit conversions", and "Explicit conversions". It explains that type conversion creates ambiguous situations and shows code examples with handwritten annotations. The code example is identical to the previous slide:

```
double f (int i, int j) { return (double) i / j; }

// compiler generates:
double f (int i, int j) {
    double temp_i = i, temp_j = j; // Conversion in temporary
    return temp_i / temp_j;
}
```

Annotations include:
i : unk
2
int
double
3
int

For the simple reason that, thinks about a case of i being integer, it has a certain representation. So, this two has a representation which is an integer representation, which typically is a single field representation where you use some kind of a particular way to show the signed numbers. So, you use a particular style for doing that. And when you take it to double, then you get into a different representation which is a usually multipart, where you have a sign explicitly put in terms of whether it is positive or negative then you keep a certain characteristics of this and you keep a mantises part of it.

(Refer Slide Time: 09:36)

Type Casting

- Why casting?
 - Casts are used to convert the type of an object, expression, function argument, or return value to that of another type
- (Silent) Implicit conversions
 - The standard C++ conversions and user-defined conversions
- Explicit conversions
 - Type is needed for an expression that cannot be obtained through an implicit conversion more than one standard conversion creates an ambiguous situation
 - To perform a type cast, the compiler
 - Allocates temporary storage
 - Initializes temporary with value being cast

```
double f (int i,int j) { return (double) i / j; }

// compiler generates:
double f (int i, int j) {
    double temp_i = i, temp_j = j; // Conversion in temporary
    return temp_i / temp_j;
}
```

Handwritten note: $35.7 \equiv 0.357 \times 10^2$

Diagram illustrating floating-point representation:

Binary representation of 35.7: 0.10110011×10^2

So, all that you say that if my number is 35.7, then I do not represent it; let me clear that (Refer Time: 09:36) if my number is 35.7, then I do not represent it as 35.7 or anything like that. I represent it in some normalize form, typically what the systems do these days that you convert it to 0.357 into 10 to the power 2, may be you will not do 10 to the power 2, you will possibly do this power in binary. I am just explaining it in decimal. So, the representation then becomes that my number is positive, my fractional part is 0.357 that is it is a fraction which is less than 1 and then my power is 2. So, this representation gives me 35.7.

Now, certainly if I want to take it to integer, this will become 35, which is your 10s complement, but or 2s complement representation which is very different. Even if this

were 35.0, this was 0.350, this representation and the representation of 35 is very different. So, what is the consequence of that it is that you cannot just wish that through casting the type has changed, you actually have a different kind of value when you have changed it.

(Refer Slide Time: 11:01)

The screenshot shows a presentation slide titled "Type Casting". The slide content is as follows:

- Why casting?
 - Casts are used to convert the type of an object, expression, function argument, or return value to that of another type
- (Silent) Implicit conversions
 - The standard C++ conversions and user-defined conversions
- Explicit conversions
 - Type is needed for an expression that cannot be obtained through an implicit conversion more than one standard conversion creates an ambiguous situation
- To perform a type cast, the compiler
 - Allocates temporary storage
 - Initializes temporary with value being cast

```

double f (int i,int j) { return (double) i / j; }

// compiler generates:
double f (int i, int j) {
    double temp_i = i, temp_j = j; // Conversion in temporary
    return temp_i / temp_j;
}

```

Handwritten annotations on the slide:

- A red box highlights the line "double f (int i, int j) { return (double) i / j; }". Inside this box, there is a small red "i" above the first "i" and a red "d" above the first "d" in "double".
- A red arrow points from the handwritten "i" to the handwritten "double(i)".
- Red arrows point from the handwritten "temp_i" and "temp_j" to the handwritten "Memory Computation".

So, if I have i represented somewhere, and once I have cast it to $double$ that same location cannot actually carry that values. So, I have a different location some location t_1 , where I have the value of i equivalent value of i in the $double$ representation. So, in many a times, we will see that compiler actually needs to allocate separate storage for this cast value and then initialize that with the casting process.

So, cast includes in many cases not always though, I will highlight where it will not need this it, but in many cases it needs to allocate memory where the temporary value, the cast value will be put, and it needs to perform computation to take the original value, and compute the value that have cast to. So, that is the basic idea of casting. And we will need to see how that happens well.

(Refer Slide Time: 12:05)

The screenshot shows a presentation slide titled "Type Casting". The slide has a blue header bar with the title. Below the header, there is a sidebar on the left containing navigation links: "Module 32", "Partha Pratim Das", "Objectives & Outline", "Casting", "Upcast & Downcast", "Cast Operators", "cast, cast", and "Summary". The main content area contains a bulleted list under the heading "Why casting?":

- Casts are used to convert the type of an object, expression, function argument, or return value to that of another type

Under the heading "(Silent) Implicit conversions":

- The standard C++ conversions and user-defined conversions

Under the heading "Explicit conversions":

- Type is needed for an expression that cannot be obtained through an implicit conversion more than one standard conversion creates an ambiguous situation

Under the heading "To perform a type cast, the compiler":

- Allocates temporary storage
- Initializes temporary with value being cast

Below this, there is a code snippet:

```
double f (int i,int j) { return (double) i / j; }
```

Handwritten annotations highlight the line "return (double) i / j;" with a red bracket and the word "Conversion". Below the code, another snippet is shown:

```
// compiler generates:  
double f (int i, int j) {  
    double temp_i = i, temp_j = j; // Conversion in temporary  
    return temp_i / temp_j;  
}
```

Annotations here highlight "temp_i = i" and "temp_j = j" with red brackets and the words "Conversion in temporary".

At the bottom of the slide, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das". The slide number "4" is also visible.

So, if I have a function like this then in all likelihood, the compiler actually is implemented a function something like this, this remains same. But it needs to do the conversion, so it defines a new variable `temp_i`. And it is doing a conversion here, this is where the conversion is happening, this is like you can think of this as if like a constructor, so as if `double` as a constructor which is being initialized with an integer value `i`. So, `temp_i` will become a new double variable having the value which is closest to the integer value `i`.

Similarly `temp_j`, this will also need conversion will have a double corresponding to `j` and then you actually divide these two double numbers and give the result in double. So, this conversion, this conversion are what is required and these are the two fields which are required for the temporary storage. So, this is the scenario of casting that will happen in significantly in C, and that is what we keep on seeing when while we either use implicit casting, implicit conversion or when we use C style conversion in the process.

(Refer Slide Time: 13:12)

The screenshot shows a presentation slide titled "Casting: C-Style RECAP (Module 26)". The slide content includes:

- Casting is performed when a value (variable) of one type is used in place of some other type
- int i = 3;
double d = 2.5;

double result = d / i; // i is cast to double and used
- Casting can be implicit or explicit
- int i = 3;
double d = 2.5;

double *p = &d;

d = i; // implicit

i = d; // implicit -- // warning C4244: '=' : conversion from 'double' to 'int',
// possible loss of data
i = (int)d; // explicit

i = p; // error C2440: '=' : cannot convert from 'double *' to 'int'
i = (int)p; // explicit

At the bottom right, there is a video player showing a person's face, identified as Partha Pratim Das.

So, just to quickly recap we had seen a lot of C style conversion in module 26 before discussing getting into the dynamic binding. So, we have seen the usual you know integer double kind of conversion we have seen different other kinds of conversion also like, we have seen that implicit conversion is not possible in certain cases like a pointed to double cannot be converted to int, but we explicit c style casting those things can be done.

(Refer Slide Time: 13:44)

Casting: C-Style
RECAP (Module 26)

- Implicit Casting between unrelated classes is not permitted

```
class A { int i; };
class B { double d; };

A a;
B b;

A *p = &a;
B *q = &b;

a = b; // error C2679: binary '=' : no operator found
        // which takes a right-hand operand of type 'main::B'

a = (A)b; // error C2440: 'type cast' : cannot convert from 'main::B' to 'main::A'

b = a; // error C2679: binary '=' : no operator found
        // which takes a right-hand operand of type 'main::A'

b = (B)a; // error C2440: 'type cast' : cannot convert from 'main::A' to 'main::B'

p = q; // error C2440: '=' : cannot convert from 'main::B *' to 'main::A *'

q = p; // error C2440: '=' : cannot convert from 'main::A *' to 'main::B'

p = (A*)ab; // Forced -- Okay
q = (B*)aa; // Forced -- Okay
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

We will see what are the consequences of that we saw now the same rule if it is extended to C++ where classes come into the play, then we have seen that if there are two unrelated classes then mostly, no kind of implicit conversion between the objects are actually possible. But you can still use the C style of force casting to cast between the different pointers of these types.

(Refer Slide Time: 14:14)

Casting: C-Style
RECAP (Module 26)

- Forced Casting between unrelated classes is dangerous

```
class A { public: int i; };
class B { public: double d; };

A a;
B b;

a.i = 5;
b.d = 7.2;

A *p = &a;
B *q = &b;

cout << p->i << endl; // prints 5
cout << q->d << endl; // prints 7.2

p = (A*)ab;
q = (B*)aa;

cout << p->i << endl; // prints -858993459 ----- GARBAGE
cout << q->d << endl; // prints -9.25596e+061 ----- GARBAGE
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

We have seen the danger of doing this kind of casting; and because we have seen in this example earlier below that it is possible that I have forced a casting, which is not to be cast, because it is a two unrelated classes. And the compiler allowed us to do that and then gave out certain garbage output.

(Refer Slide Time: 14:38)

Casting on a Hierarchy: C-Style:
RECAP (Module 26)

Module 32
Partha Pratim Das

Objectives & Outline
Casting
Upcast & Downcast
Cast Operators
const_cast
Summary

• Casting on a hierarchy is permitted in a limited sense

```
class A {};
class B : public A {};

A *pa = 0;
B *pb = 0;
void *pv = 0;

pa = pb; // okay ----- // UPCAST || ✓
pb = pa; // error C2440: '=>' : cannot convert from 'A *' to 'B *' // DOWNCAST ||
pv = pa; // okay ----- // Lose the type
pv = pb; // okay ----- // Lose the type

pa = pv; // error C2440: '=>' : cannot convert from 'void *' to 'A *'
pb = pv; // error C2440: '=>' : cannot convert from 'void *' to 'B *'
```

NPTEL MOOCs Programming in C++
Partha Pratim Das

So, going further on that, we saw the casting still may make some sense, if the classes are on a hierarchy. And in that, we saw that there could be up-cast or down-cast; up-cast is specialization to generalization; downcast is generalization to specialization. And we saw that up-cast is usually a relatively safe thing to do.

(Refer Slide Time: 15:02)

Casting on a Hierarchy: C-Style:
RECAP (Module 26)

- Up-Casting is safe

```
class A { public: int dataA_; };
class B : public A { public: int dataB_; };

A a;
B b;

a.dataA_ = 2;
b.dataA_ = 3;
b.dataB_ = 5;

A *pa = &a;
B *pb = &b;

cout << pa->dataA_ << endl; // prints 2
cout << pb->dataA_ << " " << pb->dataB_ << endl; // prints 3 5

pa = &b;

cout << pa->dataA_ << endl; // prints 3
// cout << pa->dataB_ << endl; // error C2039: 'dataB_': is not a member of 'A'
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, you can up-cast because when you are up-casting, you are using less information than what actually exist. So, we saw that these is up-cast are usually, OK; but if you down-cast then you are going from generalization to specialization, so you actually have less information and you are trying to infer more information and that gets risky.

(Refer Slide Time: 15:26)

Casting in C and C++

- Casting in C
 - Implicit cast
 - Explicit C-Style cast
 - Loses type information in several contexts
 - Lacks clarity of semantics
- Casting in C++
 - Performs fresh inference of types without change of value
 - Performs fresh inference of types with change of value
 - Using implicit computation
 - Using explicit (user-defined) computation
 - Preserves type information in all contexts
 - Provides clear semantics through cast operators:
 - const_cast
 - static_cast
 - reinterpret_cast
 - dynamic_cast
 - Cast operators can be grep-ed in source
 - C-Style cast must be avoided in C++

NPTEL MOOCs Programming in C++ Partha Pratim Das 10 7:00 PM 8/22/2016

So, coming to C++, the casting takes a completely different view significantly different view compared to C. In C, you had implicit cast and C style cast all of which know. And the two major factors is, it often loses the type information in the several context, because you can force something into an expression into a certain type without really bothering about what happens to the representation, and the correctness of computation. And the casting really lacks the clarity of semantics. C++ has tried making this far more uniform and far more type consistent, so that your type always remains correct. So, in C++, you can perform fresh inference of types actually often without changing the value, because you often need to infer about type where you really do not want the value to change, you do not need the value to change, and we will show examples of that.

And in other cases, while you infer the type like you were doing in case of inferring an integer as a double; you need to change the value. And this can be done either through implicit computation, where you do not need to write how the change will happen. We have not written how integer is getting change to double or it could be situations where I will need to say the user will need to define how through the casting the change will happen. And with all that, in C++, the casting preserves type information in all context that is a very very important factor which does not is not guaranteed in C. And it provides a clear semantics which is not also the case in C by making use of certain cast operators. So, there are four cast operators, which you can explicitly write out to cast one value into another.

And the reason the cast operators are highly encouraged in comparison to implicit casting or explicit C style casting is a fact that if you have cast operators, you can see they have very peculiar kind of name `const_cast`, `static_cast`, `reinterpret_cast`, `dynamic_cast`. So, if you have use them then you can easily do a grep kind of operation, grep is basically searching for a string in a file, those you have used Unix know this very well. So, you can use or you can use in your systems search operation, so if you just look for `const_cast`, you will able to see exactly what are the points in your source code where you have used the constant casting mechanism. So, it has a lot of advantages.

And anything that you can cast by C style, you can cast more meaningfully using these operators. And therefore, once you have understood the casting in C++, you should not at

all use the C style casting at all in your code. If you need to use that, you will know that there is something wrong in your design or your understanding of coding.

(Refer Slide Time: 18:27)

The screenshot shows a presentation slide titled "Cast Operators". The slide has a blue header with the title and a sidebar on the left containing navigation links for "Module 32", "Partha Pratim Das", "Objectives & Outline", "Casting", "Upcast & Downcast", "Cast Operators", "const_cast", and "Summary". The main content area contains two bullet points:

- A **cast operator** takes an expression of **source type** (implicit from the expression) and convert it to an expression of **target type** (explicit in the operator) following the **semantics of the operator**
- Use of cast operators increases robustness by generating errors in **static** or **dynamic** time

Below the text is a hand-drawn diagram illustrating casting. It shows a variable *i* of type T_1 (labeled "source") being converted to type T_2 (labeled "target"). The conversion arrow is labeled "Cast". The text "Cast < T_2 ->" is written near the arrow. There is also some handwritten text "work" with an arrow pointing to the target type T_2 .

In the bottom right corner of the slide, there is a circular video feed showing a person, identified as Partha Pratim Das, speaking. The bottom of the slide displays the text "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

Now, usually a casting is as I said that a variable is of certain type T_1 and I want to take it to some type T_2 . So, the way it works is a cast operator takes. So, this is what we say is a source type and this is what we say is a target type. So, a cast operator takes the expression and specifies what is the target type, and that is how this conversion will happen, now it does not need to specify the source type because in C++ all expressions, all variables have a fixed type at any point they have a type.

So, knowing *i*, I will know the source type T_1 because *i* is of type T_1 , but I need to know - what is a destination type, what is a target type and that will specify like this. And C++ has operators which can do casting at the static time or at the dynamic time.

(Refer Slide Time: 19:29)

The screenshot shows a presentation slide titled "Cast Operators". The slide content is as follows:

- const_cast operator: `const_cast<type>(expr)`
 - Explicitly overrides const and/or volatile in a cast
 - Usually does not perform computation or change value
- static_cast operator: `static_cast<type>(expr)`
 - Performs a non-polymorphic cast
 - Usually performs computation to change value – implicit or user-defined
- reinterpret_cast operator: `reinterpret_cast<type>(expr)`
 - Casts between unrelated pointer types or pointer and integer
 - Does not perform computation yet reinterprets value
- dynamic_cast operator: `dynamic_cast<type>(expr)`
 - Performs a run-time cast that verifies the validity of the cast
 - Performs pre-defined computation, sets null or throws exception

On the right side of the slide, there is a circular video feed showing a person's face. At the bottom left, it says "NPTEL MOOCs Programming in C++". At the bottom right, it says "Partha Pratim Das".

So, therefore, cast operators which will discuss one after other. One is a first is a const cast operator; this operator is used to basically deal with overwrite the const-ness or volatility that is the c-v qualification of an expression. So, it can convert take away the const-ness from a constant expression, it can add const-ness to a non-constant expression and so on. Static cast is done primarily through polymorphic is a polymorphic cast. So, if you are not on a polymorphic hierarchy then you can use static cast; and it often engages user defined casting, we will see that.

Reinterpret cast is something between unrelated pointer type or between pointer and integer you can to reinterpret cast which is very, very risky; it is pretty much like the c style cast and should be very sparing used. And the most important of the cast is the dynamic cast, where you do the casting at the runtime based on the runtime time. So, we will see through all of these starting from the first one.

(Refer Slide Time: 20:37)

The screenshot shows a presentation slide titled "const_cast Operator". The slide content includes:

- `const_cast` converts between types with different cv-qualification
- Only `const_cast` may be used to cast away (remove) const-ness or volatility
- Usually does not perform computation or change value

Handwritten notes on the slide include: "creat int i" and "i [5]".

The video feed in the bottom right corner shows a man with glasses and a blue shirt, identified as Partha Pratim Das.

First is a `const_cast` which is `const cast` converts between types of different c-v qualification c-v, you remember that is c stands for `const` v stands for `volatile`. So, if I have a variable declaration, I can qualify that by saying that it is either a constant that it will never change after it has been constructed or it could be volatile that is it can change at any time without our knowledge. So, whenever we have that we have the c-v qualification and `const cast` can change that is, only `const` which can take away or remove the `const-ness` or `volatility` of an expression.

And usually does not perform any computation or change the value, because `const-ness` is more not in the value `const-ness` are in our understanding of the value. See if I have a variable `i`, and I know it is `int` now the variable at present the variable `i` at present may have value 5, now whether or not this value 5 can be changed in future is not a property of the value 5. But it is understanding of or interpretation of this variable `i` as to whether it is `const`; if it is `const` then you are saying that this cannot be changed; if it is not `const` this can be changed.

(Refer Slide Time: 22:06)

- `const_cast` converts between types with different cv-qualification
- Only `const_cast` may be used to cast away (remove) const-ness or volatility
- Usually does not perform computation or change value

int i;
const int& r = i;
r = 5;
i = 6; i

NPTEL MOOCs Programming in C++ Partha Pratim Das

And we have seen earlier also while discussing const-ness as a property that I may have a non-constant variable say `i` and I may have a constant reference to this variable. It is possible that I have a constant reference to this variable, which will mean that I cannot make assignments to this reference because this is a constant one while I can actually make changes to the variable itself. So, it can be looked at a multiple different ways.

(Refer Slide Time: 23:36)

```
#include <iostream>
using namespace std;

class A { int i; 
public: A(int i) : i_(i) {} 
    int get() const { return i_; } 
    void set(int j) { i_ = j; } 
}; 
void print(char * str) { cout << str; }

int main() {
    const char * c = "sample text";
    // print(c); // error: 'void print(char *)': cannot convert argument 1
    //           // from 'const char *' to 'char *'

    print(const_cast<char *>(c));
    const A a(1);
    a.get();

    // a.set(5); // error: 'void A::set(int)': cannot convert
    //           // 'this' pointer from 'const A' to 'A'
    const_cast<A*>(a).set(5);

    // const_cast<A*>(a).set(5); // error: 'const_cast': cannot convert
    //                           // from 'const A' to 'A'
    return 0;
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

Now, let us look at how does const cast work. So, I am taking a few simple situations for example, the first situation is having a print function. I mean do not worry about the functionality, it basically takes a char* pointer and prints that. And in the application, I have a char* pointer which is a const pointer because this is constant string; sample text is a constant string.

So, if I tried to do print c, you will get an error. Why will you get an error, because this c is a constant is a pointer to the constant data, you know the point constant is on this side, so the data cannot change, but the actual parameter the formal parameter in print is a non constant pointed to a non constant data, so it can change. So, this sees that if I pass c to this function then c might get changed; therefore, this call will not be allowed. So, you will get this is one kind of possible error that you can get.

(Refer Slide Time: 23:34)

```
#include <iostream>
using namespace std;

class A { int i_; 
public: A(int i) : i_(i) {} 
    int get() const { return i_; } 
    void set(int j) { i_ = j; } 
}; 
void print(char * str) { cout << str; }

int main() {
    const char * c = "sample text";
    // print(c); // error: 'void print(char *)': cannot convert argument 1
    //           // from 'const char *' to 'char *'

    print(const_cast<char *>(c));
    const A a(1);
    a.get();

    // a.set(5); // error: 'void A::set(int)': cannot convert
    //           // 'this' pointer from 'const A' to 'A'

    const_cast<A*>(a).set(5);

    // const_cast<A*>(a).set(5); // error: 'const_cast': cannot convert
    //           // from 'const A' to 'A'
    return 0;
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

Now if you want to actually really want to call this function, then you need to strip c of its const-ness. So, you can do it in this way; const_cast is a name of the cast, c is the expression that you want to cast and this is your target type, your target type is char*. So, c was of type const char star, this was your source type. And now you are made it char star, so the type of this whole expression const_cast char* c is char*, not const char *. So, it takes the const char * type stripes of the const and gives up a char *. Now once it

becomes char * then it is same as the char* type you have. So, now, you can call this function.

(Refer Slide Time: 24:35)

The slide title is "const_cast Operator". The left sidebar shows a navigation menu with "Module 32", "Partha Pratim Das", "Objectives & Outline", "Casting", "Uecast & Downcast", "Cast Operators", "const_cast", and "Summary". The main content area contains the following C++ code:

```
#include <iostream>
using namespace std;

class A { int i; };
public: A(int i) : i_(i) {}
    int get() const { return i_; }
    void set(int j) { i_ = j; }
};

void print(char * str) { cout << str; }

int main() {
    const char * c = "sample text";
    // print(c); // error: 'void print(char *)': cannot convert argument 1
                // from 'const char *' to 'char *'

    print(const_cast<char *>(c));
    const A a(1);
    a.get();

    // a.set(5); // error: 'void A::set(int)': cannot convert
                // 'this' pointer from 'const A' to 'A'
    const_cast<A*>(a).set(5);

    // const_cast<A*>(a).set(5); // error: 'const_cast': cannot convert
                                // from 'const A' to 'A'
    return 0;
}
```

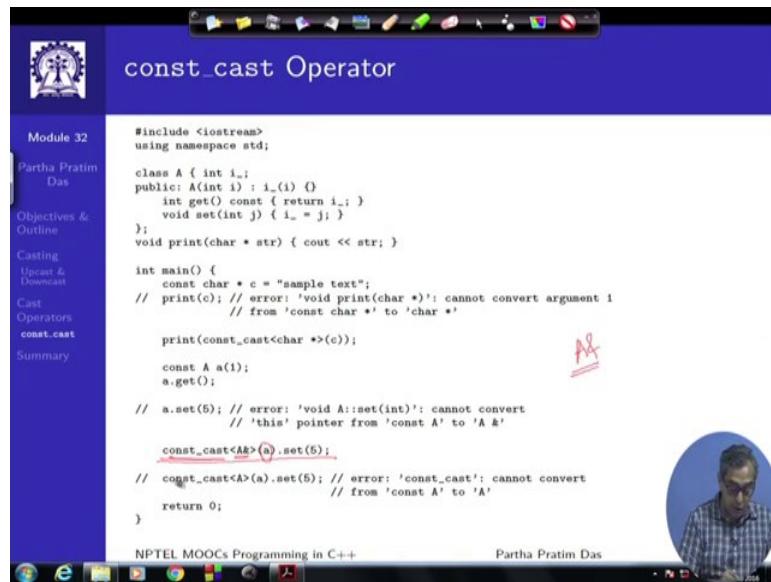
Handwritten annotations include a red arrow pointing to the "const" keyword in "const char *" with the note "A + your". Another red arrow points to the "const" keyword in "const_cast<char *>(c)" with the note "const".

With this trick you can call the function in this particular case, and because you are using const cast, anybody who reads to this code will immediately understand that you needed to strip the const-ness from c, and therefore, you have used it. Think about another situations suppose you have a class a, and that has a const member function and a non-const member, const member function cannot change the contents of the class, non-const member function can. So, if you have a constant object a then calling a.get() is fine because a.get() is a const member function it can be called with const or non const object, because it guarantees that it will not change.

Now think about calling a.set(), a.set() this is a non constant member function it actually changes the object, and you are said that a is a constant object. So, a.set() is an error, because you are you cannot be allowed to change the value of a constant object which you can if you are using a non-constant member function. In terms of type basically this pointer of a, has a type const, it has a type which is of this. So, it says that this point is a constant pointer it points to a constant a object. And therefore, but set this function is a non-constant one. So, it needs a pointer of type a * const, but you have a

pointer of this type. So, you cannot pass it there, because if you can pass it there, then it can violate change anything and go ahead.

(Refer Slide Time: 26:34)



The screenshot shows a presentation slide titled "const_cast Operator". The slide content is as follows:

```
#include <iostream>
using namespace std;

class A { int i; };
public: A(int i) : i_(i) {}
    int get() const { return i_; }
    void set(int j) { i_ = j; }
};

void print(char * str) { cout << str; }

int main() {
    const char * c = "sample text";
    // print(c); // error: 'void print(char *)': cannot convert argument 1
                // from 'const char *' to 'char *'

    print(const_cast<char *>(c));
    const A a(1);
    a.get();

    // a.set(5); // error: 'void A::set(int)': cannot convert
                // 'this' pointer from 'const A' to 'A &'

    const_cast<A&>(a).set(5);
    // const_cast<A>(a).set(5); // error: 'const_cast': cannot convert
                            // from 'const A' to 'A'
    return 0;
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, if you still need to call that function, what you can do we can strip the const-ness of the object from A and cast it, you do const cast and now you make it A&, that is a taking a reference you are creating a reference of A which is a constant object. And you are creating a non constant reference to that.

So, this resultant expression is an object which has this pointer which does not point to a constant object it points to a non-constant object now and therefore, you can it is to call the set function on this. But certainly you cannot do a similar thing using, you cannot strip the const-ness of the whole object altogether that is not allowed because that will actually mean that you need to construct a new object and do something else. So, this will still continued to be an error, but you can create a non-constant reference to the same object by doing the const cast.

(Refer Slide Time: 27:45)

The screenshot shows a presentation slide titled "const_cast Operator vis-a-vis C-Style Cast". The slide content is as follows:

```
#include <iostream>
using namespace std;

class A { int i; }
public: A(int i) : i_(i) {}
    int get() const { return i_; }
    void set(int j) { i_ = j; }
};

void print(char * str) { cout << str; }

int main() {
    const char * c = "sample text";
    // print(const_cast<char *>(c));
    printf((char *)c); // C-Style Cast

    const A a(1);

    // const_cast<A&>(a).set(5); // C-Style Cast
    ((A)a).set(5); // error: 'const_cast'; cannot convert
                    // from 'const A' to 'A'
    ((A)a).set(5); // C-Style Cast

    return 0;
}
```

The slide also includes a sidebar with navigation links: Module 32, Partha Pratim Das, Objectives & Outline, Casting, Upcast & Downcast, Cast Operators, const_cast, and Summary. The footer of the slide indicates it is from NPTEL MOOCs Programming in C++.

So, this is the basic reason you need const cast for I have put in several other examples for you to practice. For example, this one show that how does it looks like if you do a C style. For example, here we had shown that you can strip const-ness of the string and make the function call go through.

Alternatively you can use C style and do this, I will strongly advice against doing this, because if you do this then anybody reading this will understand that you are striping const-ness of. If you write this then somebody reading this does not know why you are doing this C could have been you are doing this possibly because you know want to remove const-ness, but you are doing this may be because you has a void star pointer. So, you just want to make it understand it as a char star and so on. So, C style of casting does not give you any information and should be avoided.

Similarly, you could do this by creating a non-const reference to the constant object, and make this call we just saw that. You could also do this by a C style casting of this. You can cast the object to a non-const reference and then use that. I will again strongly advice against using this, because here it is clear that you are just removing the const-ness, here it is not clear as to what we are trying to do, it is not possible to make out, it will have to really understand that.

And the most dangerous thing is even if you tried to const cast, the whole object cannot be const cast. We cannot take a constant object and const cast to a non-constant object, because that will mean a completely different object. So, this even with const cast is an error, but surprisingly with c style this is permitted. So, you are actually doing something which is kind of illegal and goes against the basic premise of const-ness that you have built up. So, these should be strongly discouraged. So, please do not use this kind of things and all.

(Refer Slide Time: 29:45)

```
#include <iostream>
using namespace std;
struct type { type() :i(3) {}
    void m1(int v) const {
        //this->i = v; // error C3490: 'i' cannot be modified because
        // it is being accessed through a const object
        const_cast<type*>(this)->i = v; // OK as long as the type object isn't const
    }
    int i;
};
int main() {
    int i = 3; // i is not declared const
    const int* cref_i = i;
    const_cast<int*>(cref_i) = 4; // OK: modifies i
    cout << "i = " << i << '\n';
    type t; // note, if this is const type t;, then t.m1(4); is undefined behavior
    t.m1(4);
    cout << "type::i = " << t.i << '\n';
}
[ const int j = 3; // j is declared const
int* pj = const_cast<int*>(j);
*pj = 4; // undefined behavior! Value of j and *pj may differ
cout << j << " " << *pj << endl;
void (type::*fp)(int) const = &type::m1; // pointer to member function
//const_cast<void(type::*)>(fp); // error C2440: 'const_cast' : cannot convert
// from 'void __thiscall type::*' (int) const' to 'void __thiscall type::*'(int)'
// const_cast does not work on function pointers
return 0;
}
NPTEL MOOCs Programming in C++ Partha Pratim Das 16 7:34 PM 4/20/2018
```

Finally, I have given some I say set of other examples here, showing you that if you have a constant member function then you can still make changes within that constant member function by changing the const-ness of this pointer within that member function. You can still do some tricks like that. Of course, do not try this with c style casting, because it will become very dangerous.

And just read through this line by line later on. I will just point out to one small piece of the code here. Here, I have defined a constant integer and initialize it with 3. I have defined integer pointer, but what I have done is have taken the address of this variable, which will be a pointer to a constant integer, because it is a constant integer. And therefore, I have striped the const-ness by the const cast.

So, I have `j` here which is constant and I have `p j` here which points which is non-constant. So, this const-ness has allowed me to create this. And since this is non-constant, I can assign through this because it is pointing to a non-constant integer. So, I can do `start p j` and assign a value. If you do this after all this you accept undefined behavior you do not know what is going to happen. For example, all that we know that pointer points to a variable then if I print the value of the variable and if I deference and print the value from the pointer, I should get the same value.

So, if you look at this carefully if I print `j` and star `pj`, `j` is a variable `pj` is the pointer to this variable, `*pj` is certainly has to be same variable value. If I print that this is output that gets generated, that this is 3 and this is 4. So, you are infer a quite a bit of surprise, if you use the const-ness to strip off, the const cast to strip of const-ness arbitrarily. The reason this happens is a pretty simple that the compiler particular compiler that have used knowing that `j` is a const and 3 has actually replace 3 in this place. So, it does not have `j` anymore, because it knows that it is const, so it cannot change and this one is a changing value.

(Refer Slide Time: 32:24)

```

Module 32
Partha Pratim Das
Objectives & Outline
Casting
Update & Download
Cast Operators
const_cast
Summary

const_cast Operator

#include <iostream>
using namespace std;
struct type { type() :i(3) {} void m1(int v) const {
    //this->i = v; // error C3490: 'i' cannot be modified because
    // it is being accessed through a const object
    const_cast<type*>(this)->i = v; // OK as long as the type object isn't const
}
int i;
};

int main() {
    int i = 3; // i is not declared const
    const int& cref_i = i;
    const_cast<int*>(cref_i) = 4; // OK: modifies i
    cout << "i = " << i << '\n';

    type t; // note, if this is const type t, then t.m1(4); is undefined behavior
    t.m1(4);
    cout << "type::i = " << t.i << '\n';

    const int j = 3; // j is declared const
    int* pj = const_cast<int*>(&j);
    *pj = 4; // undefined behavior! Value of j and *pj may differ
    cout << "j<< " << *pj << endl;

    void (type::*fp)(int) const = &type::m1; // pointer to member function
    //const_cast<void(type::*)>(fp); // error C2440: 'const_cast' : cannot convert
    // from 'void __thiscall type::> (int)' to 'void __thiscall type::> (int)'
    // const_cast does not work on function pointers
    return 0;
}

NPTEL MOOCs Programming in C++
Partha Pratim Das
16

```

So, when it had to do this conversion it has silently created another location, so `pj` actually. So, `j` is here which has three when this conversion was done `pj` actually does not

point here, but it points to a new temporary location whose value is initialized with the value of 3 and then the changes happened within that. You can clearly see that they give you different results. So, go through this, you will get further details, you can learn that function pointers const cast cannot be removed from the function pointers and so on.

(Refer Slide Time: 33:00)

The slide has a blue header bar with the title "Module Summary". On the left, there is a sidebar with a logo and navigation links: "Module 32", "Partha Pratim Das", "Objectives & Outline", "Casting", "Upcast & Downcast", "Cast Operators", "const_cast", and "Summary". The main content area contains a bulleted list of learning objectives:

- Understood casting in C and C++
- Explained cast operators in C++ and discussed the evils of C-style casting
- Studied `const_cast` with examples

At the bottom, there is a circular profile picture of Partha Pratim Das and the text "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

To summarize, we have tried to understand the basic process of casting in C and C++. And particularly explained cast operators, a basic structure of the cast operator and discuss the evils of C-style casting in a number of examples. And particularly, we have taken a look into the `const cast` operator.

In the next module, we will take up the other cast operators and move on.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 48
Type Casting and Cast Operators: Part – II

Welcome to Module 33 of Programming in C++. We have been discussing about type casting in C and V plus plus, and particularly the cast operators in C++.

(Refer Slide Time: 00:31)

The screenshot shows a presentation slide with a dark blue header bar. The title 'Module Objectives' is centered in white text. Below the header, the main content area is white. On the left side, there is a vertical sidebar with a dark blue background. At the top of this sidebar is a small logo of an open book with a circular emblem above it. Below the logo, the text 'Module 33' is displayed in white. Underneath 'Module 33', the name 'Partha Pratim Das' is listed. A horizontal line separates this from a list of topics. The list includes 'Objectives & Outline', 'Cast Operators', 'static_cast', 'reinterpret_cast', and 'Summary'. To the right of the sidebar, the main content area contains a single bullet point: '• Understand casting in C and C++'. At the bottom of the slide, there is a footer bar with a dark blue background. In the center of the footer, the text 'NPTEL MOOCs Programming in C++' is written in white. To the right of this, the name 'Partha Pratim Das' is also in white. On the far right edge of the footer bar, the number '2' is visible, indicating this is the second slide in the module.

So, we continue with that same objective of understanding casting in C and C++.

(Refer Slide Time: 00:37)

The screenshot shows a presentation slide titled "Module Outline". The slide content is as follows:

- Casting: C-Style: RECAP
 - Upcast & Downcast
- Cast Operators in C++
 - `const_cast` Operator
 - `static_cast` Operator
 - `reinterpret_cast` Operator
 - `dynamic_cast` Operator
- `typeid` Operator

At the bottom right of the slide, there is a circular video player showing a man with glasses and a blue shirt, identified as Partha Pratim Das.

In terms of the module outline, I had explained in the last module itself that will cover spread over multiple modules. So, here the blue ones that is these two are enough focus to discuss in the current module - the static cast operator and the reinterpret cast operator.

(Refer Slide Time: 00:57)

The screenshot shows a presentation slide titled "Casting in C and C++". The slide content is as follows:

- Casting in C
 - Implicit cast
 - Explicit C-Style cast
 - Loses type information in several contexts
 - Lacks clarity of semantics
- Casting in C++
 - Performs fresh inference of types without change of value
 - Performs fresh inference of types with change of value
 - Using `implicit` computation
 - Using `explicit` (user-defined) computation
 - Preserves type information in all contexts
 - Provides clear semantics through cast operators:
 - `const_cast`
 - `static_cast`
 - `reinterpret_cast`
 - `dynamic_cast`
 - Cast operators can be `grep`-ed in source
 - C-Style cast must be avoided in C++

At the bottom right of the slide, there is a circular video player showing a man with glasses and a blue shirt, identified as Partha Pratim Das.

We have looked at the basic issues of a casting in C and C++. We have noted that C has

the two primary style of implicit and explicit C style casting, and these are the two major issues, it often losses the type information and lacks the clarity of semantics. In contrast, C++ preserves all that; and we will see that how it can introduce user defined computation through casting, and these are all done through cast operators. And we have taken a look into the one type of const cast operator which significantly falls in this category that in most cases when you do const cast. You are just making a fresh inference about the type it was some const used to exist in an object you are trying to create a reference which is a non-constant one. Or some constant was not there, which we are trying to add through a reference or through a pointer, but usually you do not add a new computation of values, but we will see in the next type of casting, static casting, that actual explicit user defined computation will come into play.

(Refer Slide Time: 02:08)

static_cast Operator

Module 33
Partha Pratim Das
Objectives & Outline
Cast Operators
static.cast
reinterpret.cast
Summary

- `static.cast` performs all conversions allowed implicitly (not only those with pointers to classes), and also the opposite of these. It can:
 - Convert from `void*` to any pointer type
 - Convert integers, floating-point values and enum types to enum types
- `static.cast` can perform conversions between pointers to related classes:
 - Not only up-casts, but also down-casts
 - No checks are performed during run-time to guarantee that the object being converted is in fact a full object of the destination type
- Additionally, `static.cast` can also perform the following:
 - Explicitly call a single-argument constructor or a conversion operator
 - The User-Defined Cast
 - Convert to rvalue references
 - Convert enum class values into integers or floating-point values
 - Convert any type to void, evaluating and discarding the value

NPTEL MOOCs Programming in C++
Partha Pratim Das

So, with that, we get start started with the static cast operator. The first thing about static cast operator, is first of all it is static cast, that names tells us that this deals with casting which can be decided a compile time, anything that is static means it is static time. So, it is something that you can decide at the compile time. And the static cast performs or covers all conversions that are allowed implicitly, not only to those relating to pointer their reverse as well as conversion of void start to any pointer type or all of these different kinds of you know built in conversion, convert between integers, integer

floating point, enum types to enum types, integers enum types and so on all these can be done through the static cast operator.

Second, it can further perform conversion between pointers to related types that is pointers which point to two different classes on a hierarchy. And it can not only do up-cast, but it can also do down-cast, of course, at a risk. So, the risk is what is stated next, when you up-cast you know that you are always safe, because you have more information and you are looking at only the base part of it restricted part of this. When you down cast, the risk is that your actual pointer, your actual object may not be a specialized object, it may not have the additional information that you expect in terms of the specialized class, but you were still down casting. So, static cast run through that risk and should not normally be used for down casting, because it does not check at the run time that whether you appropriately have an object which is of the specialized type, but it still allows you to do that cast.

Further and that this is the most interesting thing is the static cast actually can explicitly call a single argument constructor or a conversion operator. So, you can call a single argument constructor or a conversion operator; and certainly since it can call these, these are codes that the user can write. So, static cast can often be used to actually cast through user-defined conversion routines; besides that it can convert the r-value references, enum to integer and so on so which we have.

(Refer Slide Time: 04:59)

The screenshot shows a presentation slide titled "static_cast Operator: Built-in Types". The slide content is as follows:

```
#include <iostream>
using namespace std;

// Built-in Types
int main() {
    int i = 2;
    double d = 3.7;
    double *pd = &d;

    i = d;           // implicit -- warning
    i = static_cast<int>(d); // static_cast -- okay
    i = (int)d;      // C-style -- okay

    d = i;           // implicit -- okay
    d = static_cast<double>(i); // static_cast -- okay
    d = (double)i;   // C-style -- okay

    i = pd;          // implicit -- error ✓
    i = static_cast<int>(pd); // static_cast -- error ✗
    i = (int)*pd;    // C-style -- okay: RISKY: Should use reinterpret_cast

    return 0;
}
```

The slide has a sidebar with navigation links: Module 33, Partha Pratim Das, Objectives & Outline, Cast Operators, static_cast, reinterpret_cast, and Summary. The footer of the slide indicates it is from NPTEL MOOCs Programming in C++.

So, let us get started, let us look into some of the simple examples first. So, these are the three variables that we have defined an integer, a double, and a pointer to double. And here we are doing a double to integer conversion. Double to integer conversion, this is an implicit one which goes through, but it gives you a warning. Why does it give you a warning, because an integer has a smaller representation size than double, so if you convert from double to integer, there is quite possible that you are losing some information, it still allows that, but you give you a warning that is the specification of the language?

But if you do it through a static cast it is ok, it does not give you any warning. Because if you are using static cast then the compiler knows that you are aware you are matured enough and you are aware that if you are taking a double to an integer, you are losing information and you that is exactly what you want to do because you have specified that. Interestingly C style cast also does the same thing, except that you just do not know what was intended here it is clear as to what is intended. The cast others direction integer to double, all things are same except for the implicit case, you do not have a warning because double as a bigger size. So, it any integer can always be represented as a double. So, you do not get the warning.

Look at another case of a trying to convert this pointer to double into integer. So, if you try to do that the implicit cast gives you an error, which is what it should, because you have a pointer, what do you mean by putting it to an integer. So, there should not be any conversion, the implicit. Static cast also gives you an error, it says that you cannot do this right, so that is pretty clear because there is no sense of taking a pointer and thinking of it as an as an integer. But if you do C style, it allows that. So, you can see C style cast can allows you almost anything and everything, and therefore, you run a major risk if you use the C style casting. We will see in the later on that actually in this context, if you really need to do this cast then you should use something like a reinterpret cast and not the C style cast and the static cast of course, will not work in this case. So, this is the simple cases of casting between the built in types.

(Refer Slide Time: 07:25)

The slide title is "static_cast Operator: Class Hierarchy". The left sidebar lists "Module 33", "Partha Pratim Das", "Objectives & Outline", "Cast Operators", "static_cast", "reinterpret_cast", and "Summary". The main content area contains the following C++ code:

```

#include <iostream>
using namespace std;

// Class Hierarchy
class A {};
class B : public A {};

int main() {
    A a;
    B b;

    // UPCAST
    A *p = &b;           // implicit -- okay
    p = static_cast<A*>(&b); // static_cast == okay
    p = (A*)&b;        // C-style -- okay

    // DOWNCAST
    q = &a;             // implicit -- error
    q = static_cast<B*>(&a); // static_cast == okay: RISKY: Should use dynamic_cast
    q = (B*)&a;         // C-style == okay

    return 0;
}

```

Annotations include red arrows pointing from the class names A and B to their respective memory locations, and a circled note "RISKY: Should use dynamic_cast" next to the static_cast line in the downcast section.

So, let us move on and try to look at the casting in terms of a class hierarchy. So, I have a put a simple class hierarchy B is A, this is a hierarchy. So, I have two objects a and b. And I can take the address of b and put it to pointer p which is of type A. So, since B is A, so this is where the object b exist and this is where the pointer p exist. So, if I am doing it like this, then I am a doing up-cast. I am going from the specialized here to the generalized here so that is as we saw earlier, this is allowed, and so implicit is ok. I can do the same thing by a static cast, here I take the address of b as an expression and target

is s star, I can do that same thing using C style, there is no a no addition, but certainly we will not encourage doing that because again we do not understand what is going on.

Let us try to look at the down cast that is B is A, and I have A, I have an A object. And q I think the declaration for q is missed out, q should be B *. So, I have a q which is of B star, so this is a pointer to B *. So, we are trying to do this. Now naturally as an implicit, this is an error, because this means that I am going from a specialized a generalize to a specialized object. So, there would be certain things missing. So, this is an error. In static cast, interestingly this is ok, why is it, because the compiler reasons that, since you have taken a address of A type object and you are trying to put it to a B type pointer you know that you are going down in the hierarchy. So, you are taking responsibility of doing this by saying that you are doing a static cast. So, the compiler will allow that, but again I would strongly, strongly, strongly advice that do not do this, and use what we will discuss as dynamic cast. Of course, C style casting will work in almost every case. So, this is what will happen if you are on a class hierarchy.

(Refer Slide Time: 09:51)

The slide is titled "static_cast Operator: Pitfall". It features a hand-drawn diagram on the right side. The diagram shows three windows: a large one labeled "Window", a smaller one labeled "SpecialWindow", and a third window partially visible. Red arrows point from the text in the slide to these windows. One arrow points to the text "Slices the object, creates a temporary and calls the method!" and another points to the "SpecialWindow" window. The slide contains the following C++ code:

```
class Window { public: virtual void onResize(); ... }
class SpecialWindow: public Window // derived class
public:
    virtual void onResize() { // derived onResize impl;
        static_cast<Window*>(*this).onResize(); // cast *this to Window,
                                                // then call its onResize;
                                                // this doesn't work!
        ... // do SpecialWindow-specific stuff
    }
    ...
};

class SpecialWindow: public Window // derived class
public:
    virtual void onResize() { // derived onResize impl;
        Window::onResize(); // Direct call works
        ... // do SpecialWindow-specific stuff
    }
    ...
};
```

Annotations in red text on the slide include:

- "Slices the object, creates a temporary and calls the method!"
- "Window" with a red arrow pointing to the first window in the diagram.
- "SpecialWindow" with a red arrow pointing to the second window in the diagram.

Now I will just to show you something simple, if you use this kind of things and kind of pitfalls that you might get into. Think about that you have a class window, and you have a class special window which specializes from window. And let say window has a

method on resize, so that basically is the functions if you resize the window this function should be called, and this is a virtual function, so it can be called. Now the special window decides to override the on resize function and implement it again. And in the override, what does it want to do, it first wants to call the function of the base class, so that whatever the resizing operation a general window needs to do that gets done and then it does the special window specific stuff. This is the basic intended design.

Now, in terms of doing that, whatever did a programmer do, the programmer wrote the code like this that you are in the special window. So, this pointer points to a special window. So, it took that object cast it to window. So, it is simply cast that object to window. So, which you will say, it is absolutely fine because this has more information, I am just casting it to this and then you call resize on that. Now the point is, this will not work, this will not work. This will lead to what is known as slicing the object, because what happens is when you try to, here what you are allowed to do very freely as an up-cast is cast the pointer or cast the reference, but what this person has done is cast the whole object has a whole. Now naturally, you cannot take a special window object, and make it a window object. So, what you will do when you have this kind of a cast call this leads to having a new window object. So, you have some new window object from the current special window object. This is a temporary, so this temporary object gets created through this process. So, what happens, here this was your total special window that had a window base and that window base has been copied to w.

And then what happens, so this is now it has become another w, where it this has got copied because you have tried to do this casting. And then you have call on resize, so that resize gets called on this w, not on the original special window object, because it is temporary has got created. Remember, when I discussed about casting in the earlier module, I highlighted that the casting will create temporary objects if the compiler needs to do that, here the compiler needs to do that because it is not a pointer which it can just think is of a different type. But here you are actually asking for a different object, so that object needs to resize. So, it is created an object. It is created from the base class part of your object. So, all those values are all correct, but what is different is in the process of doing that this has w has become a different object now. And so on the size happens this and then you come back and do the special window specific stuff on the original object.

So, this part is known as slicing. You have sliced the base part out.

We talked about slicing problem in a different context, when we talked about a virtual destructor. The need for virtual destructor, but you can get into slice in through improper casting as well. So, you need to be careful about that, and just you know highlight the way you could do this is you do not need to do anything sophisticated; all that you need to do is to actually explicitly call the on resize function of the window class. So, within this, you just call this it already has this pointer of the special window. So, if you call this, this function that is this function will get called, and what is this pointer it is using it is using the this pointer of the special window object, because that is the this pointer it has.

So, in this process, what will happen, you will not actually cast the object and create a temporary, but you are actually casting that this pointer through an implicit cast which is an implicit up-cast. So, which as we know is valid and you are still referring to the same object. So, this is just to highlight that, I mean casting on the face of it looks you know something very straight forward something which is pretty safe and so on, but it could really get you into nasty problems. So, you should be careful about that.

(Refer Slide Time: 14:56)

The slide is titled "static_cast Operator: Unrelated Classes". It features two side-by-side code snippets. The left snippet is for "Un-related Types" and the right is for "B ==> A". Both snippets include declarations for classes A and B, and a main() function. Handwritten annotations in red explain the casting process:

- A handwritten arrow points from the left snippet's `a = static_cast<A>(b);` to the right snippet's `a = static_cast<A>(b);`, with the text "operator=(B)" written next to it.
- Another handwritten arrow points from the left snippet's `a = (A)b;` to the right snippet's `a = (A)b;`, with the text "operator=(B)" written next to it.
- Red arrows also point from the left snippet's `a = b;` and `a = (A)b;` to the right snippet's `a = b;` and `a = (A)b;`.
- Handwritten text "A ==> B" is written above the first set of annotations, and "B ==> A" is written above the second set.

```
#include <iostream>
using namespace std;

// Un-related Types
class B;
class A {
public:
    ~A();
};

class B {};

int main() {
    A a;
    B b;
    int i = 5;

    // B ==> A
    a = b; // error
    a = static_cast<A>(b); // error
    a = (A)b; // error

    // int ==> A
    a = i; // error
    a = static_cast<A>(i); // error
    a = (A)i; // error
}

return 0;
}

#include <iostream>
using namespace std;

// Un-related Types
class B;
class A {
public:
    A(int i = 0) { cout << "A::A(i)\n"; }
    A(const B&) { cout << "A::A(B&)\n"; }
};

class B {};

int main() {
    A a;
    B b;
    int i = 5;

    // B ==> A
    a = b; // Uses A::A(B&)
    a = static_cast<A>(b); // Uses A::A(B&)
    a = (A)b; // Uses A::A(B&)

    // int ==> A
    a = i; // Uses A::A(int)
    a = static_cast<A>(i); // Uses A::A(int)
    a = (A)i; // Uses A::A(int)
}

return 0;
}
```

So, this is another example of using static cast and the difference is I am now talking about unrelated classes. So, again I have class A and B, but the class A and B are not related, they are not on a hierarchy. And I am trying to convert an object to a b object. So, I this is a object, this is b object, so I am trying to assign b to a. I try that implicit conversion is an error, because if I do a assign b, then as we have understood through operator overloading the compiler looks for a dot operator assignment b, so it expects that this class will have an operator assignment, which takes a B type of object which does not exist. Therefore, the implicit does not work; this is gone. You try static cast you get an error; even you try you get desperate, and even try C style casting, you get an error.

Similarly, say you want to convert from integer to A type, integer is a built in type, you want to convert that to A. So, you try this, try this, try this, all of these are errors. So, what you need to do is here, what you are saying is basically, when you have two unrelated types, and you want to convert one to the other, certainly what you are saying is I have an object or an expression, and from that I need to create the an object of the target type. So, I have an object of type B, I need to use that to create an object of type A. I have an object of type int, I need to use that to create an object of type A.

If you are on a hierarchy, then you already have the object. So, it is more like you whether you are looking at it as a specialized one or a generalized one is a question, but you have some object to deal with, but here you just do not have the A object, where do you get the i object, so you need to construct that. So, it simply turns out that if you have to allow this then all that you need to provide is provide a constructor for A, which takes a B object is as simple as that. So, if you provide that then all this three become valid. What it does is you do a assign b, it checks that do I have a way to create an A object using the B object, it finds that there is a constructor, so it does that.

(Refer Slide Time: 17:35)

The slide title is "static_cast Operator: Unrelated Classes". It features two columns of C++ code. The left column shows code where type B is converted to type A. The right column shows code where type A is converted to type B. Handwritten notes in red ink explain the conversion process:

- "B => A" is written above the first column of code.
- "A => B" is written above the second column of code.
- "B' => (B prime is A)" is written next to the first column's code.
- "A = B'" is written next to the second column's code.
- "B' = B" is written next to the first column's code.
- "A = static_cast<A>(b); // Uses A::A(Bk)" is written next to the second column's code.
- "a = (A)b;" is written next to the second column's code.
- "B' = b;" is written next to the first column's code.
- "a = static_cast<A>(b); // Uses A::A(Bk)" is written next to the second column's code.
- "a = (A)b;" is written next to the second column's code.

So, it basically creates an it takes the b creates b primed object of type A, and then actually assigns b prime to a using the free copy assignment operator that A class has, A class is not defined any copy assignment operator. So, it makes use of that in the next stage to actually achieve the task. So, it is the similar thing is done if you do static cast, if you use C style casting. If you look into int providing this, then what do you need by the same logic you need a constructor of A which takes int as a parameter, so you add a constructor here. And so this if you do this, it is simply turns out that it takes i, constructs an a object through this, and after construction it simply does a copy assignment through the free copy assignment operation.

So, I have shown here what are the functions that will get called if you just trace the output you will see that the actually those functions are getting called. So, all these three forms of implicit casting, casting with static cast, or C style casting, all of them will give the same behavior. So, this is where we see that if the classes are unrelated, and in several other contexts, we can actually have a user-defined conversion this is what is user-defined conversion taking place. And one way to make the user-defined conversion take place is to actually provide an appropriate constructor for the target type from the source type.

(Refer Slide Time: 19:25)

The screenshot shows a presentation slide titled "static_cast Operator: Unrelated Classes". The slide content is as follows:

```
#include <iostream>
using namespace std;

// Un-related Types
class B;
class A { int i; public:
    operator +() { ... } // handwritten note: operator +()
};

class B { public:
    operator int() { ... } // handwritten note: operator int()
};

int main() {
    A a; B b; int i = 5;
    // B ==> A
    a = b; // error
    a = static_cast<A>(b); // error
    a = (A)b; // error

    // A ==> int
    i = a; // error
    i = static_cast<int>(a); // error
    i = (int)a; // error
}

return 0;
}
```

The right side of the slide shows the same code with additional annotations:

```
#include <iostream>
using namespace std;

// Un-related Types
class B;
class A { int i; public:
    A(int i = 0) : i_(i)
    { cout << "A::A(" << i_ << "\n"; }
    operator int()
    { cout << "A::operator int()\n"; return i_; }
};

class B { public:
    operator A() { cout << "B::operator A()\n"; return A(); }
};

int main() {
    A a; B b; int i = 5;
    // B ==> A
    a = b; // B::operator A()
    a = static_cast<A>(b); // B::operator A()
    a = (A)b; // B::operator A()

    // A ==> int
    i = a; // A::operator int()
    i = static_cast<int>(a); // A::operator int()
    i = (int)a; // A::operator int()
}

return 0;
}
```

Handwritten notes include circled annotations around the class definitions and operator overloads, and arrows pointing to specific parts of the code.

But that the story does not end there, there is something more interesting than this here in the next slide, again those two same classes. Again we are trying to take B object and cast it to an A type object there is nothing different. So, we will turn all of them will be error. But in the second case, I am trying to do something different earlier I try to take an int and construct an A object. Now, given any object I want to make it to an int, I am doing the other way round. Certainly, all of them are errors, because I have no way of knowing that given an A object how should I interpret it as an integer int i. So, all of these are errors.

How do we solve the problem, in case of this, we have seen that we could have used a constructor in A, which takes a B type parameter? And alternate way to do that is to the operator form, conversion operator this is known as a conversion operator. So, class B could write a conversion operator which you write as operator and the target type. So, this is a special operator, which will get invoke, if you are trying to do a casting of a B type object to an A type object. This operator is specifically written for that purpose and this is called the conversion operator. And the conversion operator has a very interesting style if you recall operator overloading like you wrote operator +, and whatever those types, and then you say that this gives me an A object, this I am taking an A reference and these are return type and so on.

Here you see that in this the name of the operator is operator A, the type name, because it has to change a type. So, this does not take a symbol, it takes a type as the name of the operator, it changes to A. And then it does not have a return type, it does not have a return type here, why does not it have a return type, because you are defining a conversion operator to take a B object and convert it to an A object. So what it must return it must return an A type object, it cannot return anything else. You cannot write that the return type here is int that will be stupid. The return type will have to be A, because it is converting to A. And therefore, it is super flow us to write this, and the syntax does not allow it to be return. So, this is a conversion operator.

So, where I take... so you can do write the whole logic that will you need to construct an A object from the B object, and put it down here. I have done something very ((Refer Time: 22:13)) I have just constructed a default A object and return that, but you could use the data members of B and actually construct this. So, once you provide this, again equivalently all of these become valid, but now there is no constructor in a which takes a b object rather there is a conversion operator in B, for the A types which will get used. What is interesting is how do we solve this problem.

(Refer Slide Time: 22:51)

```

Module 33
Partha Pratim Das
Objectives & Outline
Cast Operators
static_cast
Summary

static_cast Operator: Unrelated Classes

#include <iostream>
using namespace std;

// Un-related Types
class B;
class A { int i_; public:
    int i() { return i_; }
};

class B { public:
    int i() { return i_; }
};

int main() {
    A a; B b; int i = 5;
    // B ==> A
    a = b; // error
    a = static_cast<A>(b); // error
    a = (A)b; // error

    // A ==> int
    i = a; // error
    i = static_cast<int>(a); // error
    i = (int)a; // error
    return 0;
}

#include <iostream>
using namespace std;

// Un-related Types
class B { int i_ = 0; public:
    A(int i = 0) : i_(i) { cout << "A:" << i_ << endl; }
    operator int() { cout << "A::operator int()\n"; return i_; }
};

class B { public:
    operator A() { cout << "B::operator A()\n"; return A(); }
};

int main() {
    A a; B b; int i = 5;
    // B ==> A
    a = b; // B::operator A()
    a = static_cast<A>(b); // B::operator A()
    a = (A)b; // B::operator A()

    // A ==> int
    i = a; // A::operator int()
    i = static_cast<int>(a); // A::operator int()
    i = (int)a; // A::operator int()
    return 0;
}

```

Now think about, how could you have solve this problem using a previous style, one way

could have been that if I have my int type, then I could have written a constructor of int which takes A type object that will convert A into int. Given A, it will give me int, exactly the way we did the previous case. Now certainly that option does not exist, because we cannot write a constructor for int, it is a built in type. So, the only option remains is take the conversion operator root. So, is the only way you can write it is for class A you write an operator int you are writing this for class A.

So, what it means it means that given a class A object, I can convert it to an int using this operator int function, and that is what I have done, I have taken this, and I have been implemented as if it returns this class has an int data members. So, I have return this data member that is a specific computation choice you can do anything else, but all that you will need to do is the since it is an operator int, it will necessarily always return an int. So, there have to be returns on int which is computed from the A object that you have. So, this is using the operator conversion operator is necessary if you want to convert some user-defined type into a built in type value. For the other, you could still use the constructor; and for two user-defined types, you could either use the constructor or use the conversion operator, but certainly you cannot use both. So, once you have done this, then all these become valid; and all of them actually use this conversion operator to convert. So, this is a basic a way the static cast operator could allow you to invoke user-defined conversion code and work between unrelated classes.

(Refer Slide Time: 24:47)

The slide title is "reinterpret_cast Operator". The left sidebar shows "Module 33" and "Partha Pratim Das". The main content lists properties of the operator:

- `reinterpret_cast` converts any pointer type to any other pointer type, even of unrelated classes
- The operation result is a simple binary copy of the value from one pointer to the other
- All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked
- It can also cast pointers to or from integer types
- The format in which this integer value represents a pointer is platform-specific
- The only guarantee is that a pointer cast to an integer type large enough to fully contain it (such as `intptr_t`), is guaranteed to be able to be cast back to a valid pointer
- The conversions that can be performed by `reinterpret_cast` but not by `static_cast` are low-level operations based on reinterpreting the binary representations of the types, which on most cases results in code which is system-specific, and thus non-portable

Handwritten notes on the slide:

- A blue box encloses the first two bullet points.
- A red circle is drawn around the last bullet point.
- A blue arrow points from a variable `v` to a blue box labeled `T1`.
- A red arrow points from the blue box `T1` to a red circle labeled `T2`.

The next is what is known as reinterpret cast operator, the third kind of. The reinterpret cast operator can convert pointer of any type to pointer of another type, even of the unrelated classes. This is the most important thing. The reinterpret cast, if you have you is doing reinterpret on something, so you have a variable `v`, which has some representation some value, if this is of type `t1`, and you reinterpret cast to `t2`, then you simply look at this as if it is a `t2` object. You do not do any computation, you do not try to do anything, you just take this address and start thinking as if it is a `t2` object that is all. So, that could give you really disastrous result, for example, `t2` could mean a bigger type. So, you are looking at here, and you could be just looking at this much. So, this is something which was not there in the variable `v`, which will think that is the part of variable `v`.

So, normally reinterpret cast is a given, so that you can convert between pointer types. As you know whatever is the system, the size of the pointer is same for all types of pointers. So, this will at least not have the size issue, but it can be used also to cast two integers as well as from integer. Now when you cast it that way, you will have to make sure that your integer type is large enough, so that the pointer address can be accommodated there. So, this is a platform specific, next point, you will should this is where platform specific feature, because C language does not guarantee that your integer

size and your pointer size is related in any way. In a large number of systems, they are the same size like 32 bit machines, Intel x86 both are 4 bytes, but it is not guaranteed by the language. So, you should be careful about the specific platform specific stuff.

So, the conversions that can be performed by reinterpret cast, but not by static cast or whole based on low-level operations of the machine. And the general understanding is that if you require reinterpret cast then you must be very very sure of what you are trying to do. Because under normal circumstances, if you are done a good design in C++, you should not require reinterpret cast. I have read several books on C++, where in the whole 300, 400 page book, there may be only one or two instances of reinterpret cast as code examples and that to just to show how reinterpret cast can be used, so that is the kind of in a model that exist.

(Refer Slide Time: 27:54)

The screenshot shows a presentation slide titled "reinterpret_cast Operator". The slide is part of "Module 33" by Partha Pratim Das. The navigation bar on the left includes "Objectives & Outline", "Cast Operators", and "reinterpret.cast". The main content area contains the following C++ code:

```
#include <iostream>
using namespace std;

class A {};
class B {};

int main() {
    int i = 2;
    double d = 3.7;
    double *pd = &d;

    i = pd; // implicit -- error
    i = reinterpret_cast<int>(pd); // reinterpret_cast -- okay
    i = (int)pd; // C-style -- okay
    cout << pd << " " << i << endl;

    A *pA;
    B *pB;

    pA = pB; // implicit -- error
    pA = reinterpret_cast<A*>(pB); // reinterpret_cast -- okay
    pA = (A*)pB; // C-style -- okay

    return 0;
}
```

At the bottom of the slide, it says "NPTEL MOOCs Programming in C++". On the right side, there is a circular video player window showing a person's face, identified as Partha Pratim Das.

But certainly having said that reinterpret cast still exists because there are certain C style cast you can do which you cannot do with any of the other three cast operators. Therefore, for example, you can convert, say if you look into this, we were looking at this little earlier also, so if you look into this and this is a pointer to double, and this is an integer this is implicitly an error, but in C style casting, you could still do that. So, reinterpret cast has been given, so that for such cases of C style casting, you have a

formal cast operator to do the cast. So, all that you say that take a double pointer as you have here, and just think of it as an integer, do not try to do anything else. Now after that the risk lies with the programmer has to what is gone wrong. It can be used for a completely, these are two completely unrelated classes pointers to do unrelated classes as you can see here. And you can cast one into another through a reinterpret cast. So, this is different; this is not casting the objects which were doing by the static cast, where we could invoke the constructor or the conversion operator.

Here we are just trying to cast the pointer. So, certainly we cannot define any logic for casting the pointer is just the way to infer the type and reinterpret cast allows you to do that. And after this has being done then you can actually dereference pA and start using that objects. Of course, what is in store for you because this was a separate object, this is a separate object. What is in store for you, it is completely unpredictable. And therefore, I would strongly, strongly recommend that do not use reinterpret cast at all. If you are requiring reinterpret cast take a second look there must be some lacunae in the design, so that you are requiring it changing the design, you will find that you will able to manage with the other three types of cast operators.

(Refer Slide Time: 30:04)

The screenshot shows a presentation slide titled "Module Summary". The slide has a dark blue header bar with the title "Module Summary" and a small circular profile picture of a person. The main content area is white with a bullet point list. On the left side, there is a vertical sidebar with a dark blue background containing navigation links: "Module 33", "Partha Pratim Das", "Objectives & Outline", "Cast Operators", "static_cast", "reinterpret_cast", and "Summary". At the bottom of the slide, there is a footer bar with the text "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

So, to summarize, we have continued on the discussion of type casting in C++. And

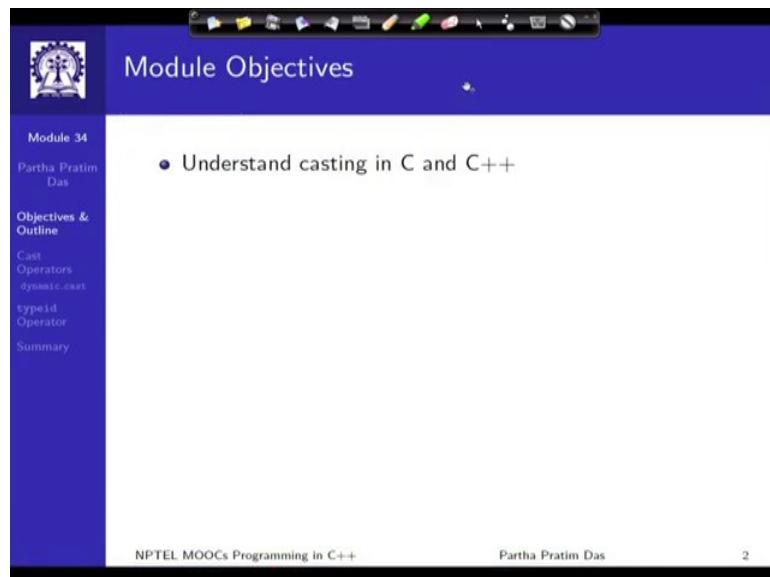
specifically, we have studied the static cast and reinterpret cast operators in this module.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 49
Type Casting and Cast Operators: Part – III

Welcome to Module 34 of programming in C++. We have been discussing about Type Casting and Cast Operators in C++.

(Refer Slide Time: 00:26)



Module Objectives

- Understand casting in C and C++

Module 34
Partha Pratim Das
Objectives & Outline
Cast Operators
dynamic_cast
typeid Operator
Summary

NPTEL MOOCs Programming in C++
Partha Pratim Das
2

So, our objective continued to be understanding casting in C and C++.

(Refer Slide Time: 00:37)

This slide shows the 'Module Outline' for Module 34, titled 'Casting: C-Style: RECAP'. The outline includes:

- Casting: C-Style: RECAP
 - Upcast & Downcast
- Cast Operators in C++
 - const_cast Operator
 - static_cast Operator
 - reinterpret_cast Operator
 - dynamic_cast Operator
- typeid Operator

The sidebar on the left lists: Module 34, Partha Pratim Das, Objectives & Outline, Cast Operators (dynamic_cast, typeid Operator), and Summary.

NPTEL MOOCs Programming in C++ Partha Pratim Das 3

As I had mentioned two modules before that this is our total outline of discussions on casting. And the blue once here the dynamic cast operator and the typeid operator is what we discussed. In the current module, we have already discussed the basic premise of casting and the three cast operators const cast, static cast, and reinterpret cast operators in C++. Our discussion on cast operators and casting will conclude with this module.

(Refer Slide Time: 01:06)

This slide is titled 'Casting in C and C++'. It compares casting in C and C++:

- Casting in C
 - Implicit cast
 - Explicit C-Style cast
 - Loses type information in several contexts
 - Lacks clarity of semantics
- Casting in C++
 - Performs fresh inference of types without change of value
 - Performs fresh inference of types with change of value
 - Using implicit computation
 - Using explicit (user-defined) computation
 - Preserves type information in all contexts
 - Provides clear semantics through cast operators:
 - const_cast
 - static_cast
 - reinterpret_cast
 - dynamic_cast
 - Cast operators can be grep-ed in source
 - C-Style cast must be avoided in C++

The sidebar on the left lists: Module 34, Partha Pratim Das, Objectives & Outline, Cast Operators (dynamic_cast, typeid Operator), and Summary.

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, this is what we had seen that C has implicit casting, and explicit style of casting. And these are the lacunae in C casting; and based on this, the cast operators are the saviors that we have. Now if we look into the cast operators, this cast operator has given us the ability to change the c v qualification.

So, the consciousness volatility view of the objects can be manipulated through const cast, particularly by using const or non-const reference to non-const or const objects or pointers constants and so on. Static cast takes care of all different kinds of implicit cast that C allows, you can explicitly write them using static cast. It has allowed us to actually do up cast as well as downcast on a hierarchy. It has also allowed us to cast between unrelated classes by using user defined constructor or conversion operators.

This is the widest form of static cast is a widest form of cast which will be used. And static cast is specifically so named, because it can do the whole inference of the casting at the compile time it does everything at the compile time. So, does const cast, but you specifically talk of static cast, as static because it is a wide variety of casting that you can do, but all of that you do based on the compile time.

Reinterpret cast as we saw is basically looking at the data through the classes of a different type and it is something which can be used to cast a pointer of any type to a pointer of another type or to cast between a pointer type and an integer type. And we have reason that we would normally avoid using the reinterpret cast.

(Refer Slide Time: 03:30)

dynamic_cast Operator

- `dynamic_cast` can only be used with pointers and references to classes (or with `void*`)
- Its purpose is to ensure that the result of the type conversion points to a valid complete object of the destination pointer type
- This naturally includes pointer upcast (converting from pointer-to-derived to pointer-to-base), in the same way as allowed as an implicit conversion
- But `dynamic_cast` can also downcast (convert from pointer-to-base to pointer-to-derived) polymorphic classes (those with virtual members) if-and-only-if the pointed object is a valid complete object of the target type
- If the pointed object is not a valid complete object of the target type, `dynamic_cast` returns a null pointer
- If `dynamic_cast` is used to convert to a reference type and the conversion is not possible, an exception of type `bad_cast` is thrown instead
- `dynamic_cast` can also perform the other implicit casts allowed on pointers: casting null pointers between pointers types (even between unrelated classes), and casting any pointer of any type to a `void*` pointer

NPTEL MOOCs Programming in C++ Partha Pratim Das

In this context, the dynamic cast is a very unique one. Dynamic cast is the only cast operator which is based on the run time behavior of the program, which is actually based on the objects that will exist at the run time. Dynamic cast does not work with objects; it is used only with pointers and references to classes. It does not work with object directly; it has to either have a pointer that you can cast or it has to have a reference that you can cast.

The purpose is to ensure that the result of type conversion points to a valid complete object of the destination pointer type; this is probably not still making much sense. So, it will come up when we go through examples, but basic point is that if the dynamic cast presumption that is a context where you bring in dynamic cast is you have polymorphic hierarchy. It is not defined, if you do not have a hierarchy; and it is ill defined on a non-polymorphic hierarchy, it has hardly any use its use is always on a polymorphic hierarchy.

So, on a hierarchy we can up cast, for which also you can use dynamic cast, but what is important is you can downcast using dynamic cast and downcast with the run time information. So, that you can actually guarantee correctness of converting a pointer to the base class to a pointer to the derived class, and really know after this conversion whether

you are pointing to a valid object or you are not pointing to a valid object.

Now what happens is if this conversion is valid that after you have come from the base class pointer to a derived class pointer, you are actually pointing to the derived class object through this pointer. Then your dynamic cast operator gives you the same value of the pointer as of the base class pointer, because certainly the address of the object cannot change. But if this conversion is not valid that is if when you have downcast, you are not actually having a derived class object, for which you want to point to then the dynamic cast will set the pointer to null, so that by checking if the pointer is null, you can figure out if it is properly downcast or not. Dynamic cast can also be used with reference type we will see the example.

(Refer Slide Time: 06:03)

dynamic_cast Operator: Pointers

```
#include <iostream>
using namespace std;

Module 34
Partha Pratim Das
Objectives & Outline
Cast Operators
dynamic_cast
typeid Operator
Summary

A
B
C

#include <iostream>
using namespace std;

class A { public: virtual ~A() {} };
class B: public A {};
class C { public: virtual ~C() {} };

int main() {
    A a; B b; C c; A *pA; B *pB; C *pC; void *pV;

    pB = &b; pA = dynamic_cast<A*>(pB);
    cout << pB << " casts to " << pA << ": Up-cast: Valid" << endl;

    pA = &a; pB = dynamic_cast<B*>(pA);
    cout << pA << " casts to " << pB << ": Down-cast: Valid" << endl;

    pA = &a; pB = dynamic_cast<B*>(pA);
    cout << pA << " casts to " << pB << ": Down-cast: Invalid" << endl;

    pA = (A*)&c; pC = dynamic_cast<C*>(pA);
    cout << pA << " casts to " << pC << ": Unrelated-cast: Invalid" << endl;

    pA = 0; pC = dynamic_cast<C*>(pA);
    cout << pA << " casts to " << pC << ": Unrelated-cast: Valid for null" << endl;

    pA = &a; pV = dynamic_cast<void*>(pA);
    cout << pA << " casts to " << pV << ": Cast-to-void: Valid" << endl;

    //pA = dynamic_cast<A*>(pV); // error: 'void *' invalid expression type for dynamic_cast

    return 0;
}

NPTEL MOOCs Programming in C++
Partha Pratim Das
```

Output:
00EFC4A8 casts to 00EFC4A8: Up-cast: Valid
00EFC4A8 casts to 00EFC4A8: Down-cast: Valid
00EFC3C4 casts to 00000000: Down-cast: Invalid
00EFC3C0 casts to 00000000: Unrelated-cast: Invalid
00000000 casts to 00000000: Unrelated: Valid for null
00EFC3D4 casts to 00EFC3D4: Cast-to-void: Valid

Let us just go ahead and start taking some example. So, what I will do is I have three classes here; class A is a base class; class B is specialized from class A. So, I have a polymorphic hierarchy. Class C is a third class, which is not related. So, we will try to illustrate what happens if you deal with unrelated class by using class C. Point to note on this hierarchy is A has a virtual destructor, which means that the hierarchy is polymorphic. This hierarchy is polymorphic, because it has a virtual function; therefore, the whole hierarchy is polymorphic. I construct three objects; I have three pointers of

three types, and `pV` is a void type pointer.

Now, let us try to start doing some tricks. So, this is `A` here, and `B` is `A`. So, if I say `pB` contains the address of `B`. So, this is a `B` object, and `pB` is a base, `B` type pointer. So, it is I have a `B` type pointer, I have pointer here, and that is what holds the object. Now if I dynamic cast `pB` to `A *` that is the pointer to `A` type object then what am I doing, what is the direction my direction is upwards, I am basically doing an up cast.

So, there should not be any problem in terms of this So, if I do that and keep it in a pointer of `A` class then I have done an Up-cast, so it should be valid. So, the result of this is if I after that I print `pB` and I print `pA`, this is a value of `pB` casts to `pA`, you can see the values are same that is basically the address of the `B` object. This is Up-cast and this is valid, first case, so that was straight forward. This is just like a base line check showing that the dynamic cast will able to cast Up-cast.

(Refer Slide Time: 08:41)

The slide also includes a logo for NPTEL MOOCs Programming in C++ and the name Partha Pratim Das.

Let us now try to downcast. So, `pA` has address of `B`. So, I have a address of `b` and I have a `b` object, and I have a `pA` which is `A` type pointer, and this holds this object here. This is possible because this is possible Up-cast. So, I can always use a base class pointer to hold a derived class object. I do a dynamic cast of `pA`, here into `B *` of this type. So,

I am doing a dynamic cast from of p A in this direction; it was of A type and now I am trying to take it to B star. So, I am doing a downcast; I am pulling it down. And the resultant I keep in pB, which is a B type pointer. What is the result?

So, after that if I print pA and pB, I find pA and pB have the same values, and the downcast is valid. What is meant by the downcast is valid, because after I have converted pB now points to this object, pB is a B type of pointer and this object is of b type, so there is no violation there is everything is in place. So, we say this downcast is a valid downcast earlier I had a type pointer and a b type object, I was holding it there. So, from p B from looking at p A, I could not have said whether the object is a b type object or is a type object. But now I have done this conversion, so I know that it is a it is an object where a valid b type object can be found. I print that all these are equal, the last part of the story.

(Refer Slide Time: 10:53)

```

Module 34
Partha Pratim Das
Objectives & Outline
Cast Operators
dynamic_cast
typeid Operator
Summary
NPTEL MOOCs Programming in C++
Partha Pratim Das
6

```

```

#include <iostream>
using namespace std;

class A { public: virtual ~A() {} };
class B: public A {};
class C { public: virtual ~C() {} };

int main() {
    A a; B b; C c; A *pA; B *pB; C *pC; void *pV;

    pB = &b; pB = dynamic_cast<B*>(pA);
    cout << pB << " casts to " << pA << ": Up-cast: Valid" << endl;

    pA = &a; pB = dynamic_cast<B*>(pA);
    cout << pB << " casts to " << pA << ": Down-cast: Valid" << endl;
    // Shows cast (A to B)

    pA = &a; pB = dynamic_cast<B*>(pA);
    cout << pB << " casts to " << pA << ": Down-cast: Invalid" << endl;

    pA = (A*)c; pC = dynamic_cast<C*>(pA);
    cout << pC << " casts to " << pA << ": Unrelated-cast: Invalid" << endl;

    pA = 0; pC = dynamic_cast<C*>(pA);
    cout << pA << " casts to " << pC << ": Unrelated-cast: Valid for null" << endl;

    pA = &a; pV = dynamic_cast<void*>(pA);
    cout << pA << " casts to " << pV << ": Cast-to-void: Valid" << endl;

    //pA = dynamic_cast<A*>(pV); // error: 'void *' invalid expression type for dynamic_cast
}


```

Output:

- 00EFFCA8 casts to 00EFFCA8: Up-cast: Valid
- 00EFFCA8 casts to 00EFFCA8: Down-cast: Valid
- 00EFFC84 casts to 00000000: Down-cast: Invalid
- 00EFFC9C casts to 00000000: Unrelated-cast: Invalid
- 00000000 casts to 00000000: Unrelated: Valid for null
- 00EFFC84 casts to 00EFFCB4: Cast-to-void: Valid

Let us do the same thing, look at here, let us do the same thing, but now let me start with pA again, but now unlike earlier case, where pA was holding the address of A type object here now it is holding the address of a type object. So, this is your pA is, it holds the address of A type object. I again try to do the same dynamic cast as I had done here exactly the same expression. So, pA being, so I am again doing a downcast, I am again

bringing it down and I will have pB now try to point to this A. So, I do this downcast here and put that pointer value to pV.

See what is the result is the third line is p A, which is address of A. And what is pB, pB is 0 – null. Why p B is 0, because it is dangerous if I can do this pointing. Because if I can do this pointing then the fact that this pointer is a p B that is of B type, I will expect that what it points to is at least a b type object, but it is not b type object it is generalized then that. So, b type object may have this much it actually has only the base part of it. So, that if you do I mean this is the basic difference between dynamic cast and static cast.

So, in this context here instead of dynamic cast, if I had done a static cast of the same expression B* and then pA, the result will be the value of pA. Because if I am doing statically, I do not know what pA is pointing to, but now that this is a dynamic cast this will able to figure out as to whether it is actually B object in that case it gives copies the value of pA to pB. Or if it is A object in which case it does not copy this value, it is a 0 to this.

So, after the dynamic cast, so and you have you have seen that in these two cases, in this case you have seen this is a second line in this case you have seen this is a third line in this case, you get a copy of that address; in this case, you get a null value. So, by simply checking if pB is null or not you know whether the dynamic cast has gone through or whether actually the pointed object is a b type object or not, so that is a basic value of the dynamic cast.

And rest of it simply rests on that, for example, I could use I can again take p A take the C type of object, and keep it is address in pA certainly these two are unrelated. Since these two are unrelated, I cannot actually put the address of the C object into pA therefore, I have forced through a C style casting there is only way I can put this address and then I tried to do this conversion of pA to C *. Certainly this conversion is not possible, so what I get is a value of this is a fourth line, what I get is a value of after the conversion into p C is a null value. So, if it is unrelated then dynamic cast will always reject. These are the special case if p A is 0, and then I try to do this cast, it will the null value, null pointers, it will always convert to null pointers.

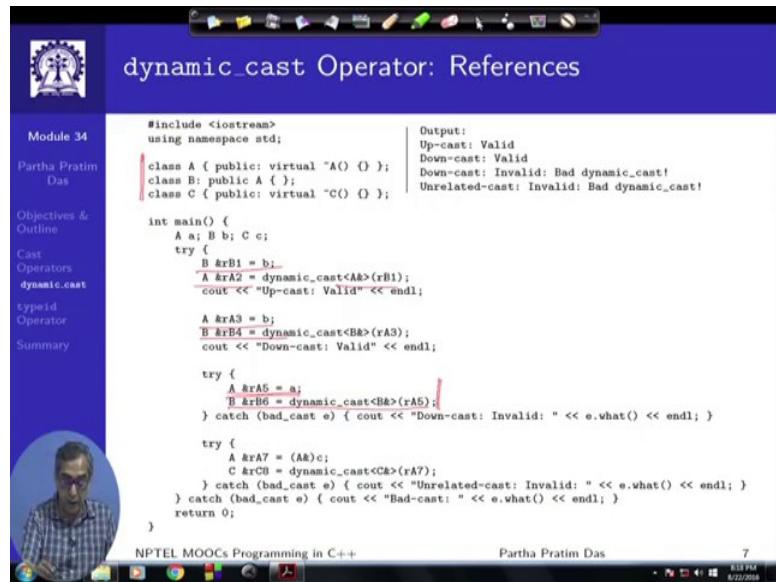
Look at this, if p A carries the address of A object, and I try to do a dynamic cast used to void star that is the void star, and this is the my void* pointer then this is a permitted cast this is a last line you can see values are same. Because it is allowed to convert from a pointer type to void, but if you try to do the reverse here you can see, I try to do the reverse, I take try to take that pV and bring it back to A star, this is compilation error.

Now in the process of doing this, in the process of doing this, certainly you will wonder as to how dynamic cast is figuring this out is very simple, is the restriction is the argument to the dynamic cast the source expression to the dynamic cast must be a pointed to a polymorphic hierarchy. What is a characteristic of polymorphic hierarchy, it is a virtual function. What is a consequence of having a virtual function, it has a virtual function pointer in that object and that pointer is specific to the type of the class as we have seen.

If I have object of a different type, I have a different virtual function table, and I have a different pointer value, but all objects of the same class same polymorphic class have the same virtual function table, and therefore, the same function pointer value. So, it basically internally takes looks at this, checks the corresponding, what is the virtual function pointer table of that object. And from that it knows that what is the type, it is dealing with.

And therefore, it can always say that whether that true dynamic type matches the static type into which you are trying to bring that address. And for that reason expressions like this cannot be expression like this that is doing a dynamic cast on pV is not possible, because void star does not is not a polymorphic type. And since void star is not a polymorphic type there is no virtual function pointer table in that. So, the dynamic cast itself, the operator itself cannot overcome this that is the reason you get invalid expression type for dynamic cast that is the basic. So, that is about the dynamic cast which can cast your pointers on a polymorphic hierarchy at the run time.

(Refer Slide Time: 17:30)



The screenshot shows a presentation slide titled "dynamic_cast Operator: References". The slide content is as follows:

```
#include <iostream>
using namespace std;

class A { public: virtual ~A() {} };
class B : public A {};
class C : public virtual ~C() {};
```

Output:
Up-cast: Valid
Down-cast: Valid
Down-cast: Invalid: Bad dynamic_cast!
Unrelated-cast: Invalid: Bad dynamic_cast!

```
int main() {
    A a; B b; C c;
    try {
        B& rB1 = b;
        A& rA2 = dynamic_cast<A&>(rB1);
        cout << "Up-cast: Valid" << endl;

        A& rA3 = b;
        B& rB4 = dynamic_cast<B&>(rA3);
        cout << "Down-cast: Valid" << endl;

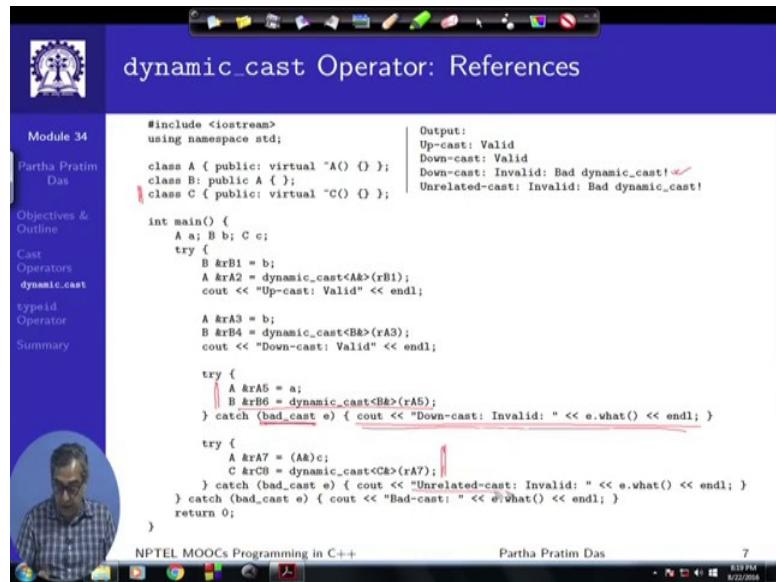
        try {
            A& rA5 = a;
            B& rB6 = dynamic_cast<B&>(rA5);
        } catch (bad_cast e) { cout << "Down-cast: Invalid: " << e.what() << endl; }

        try {
            A& rA7 = (A&)c;
            C& rC8 = dynamic_cast<C&>(rA7);
        } catch (bad_cast e) { cout << "Unrelated-cast: Invalid: " << e.what() << endl; }
        } catch (bad_cast e) { cout << "Bad-cast: " << e.what() << endl; }
    }
    return 0;
}
```

At the bottom of the slide, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

You can use that similarly with reference. So, the same hierarchy the whole everything same is such that you could now use references, and use the dynamic cast in the same way. So, this is an example of doing the up-cast, this is the example of doing the down-cast which is valid. Now, what happens if you try to do this; that you are actually having a reference of the base class object and you are trying to downcast it to the specialized class.

(Refer Slide Time: 18:25)



The screenshot shows a presentation slide titled "dynamic_cast Operator: References". The slide content is as follows:

```
#include <iostream>
using namespace std;

class A { public: virtual ~A() {} };
class B : public A {};
class C { public: virtual ~C() {} };

int main() {
    A a; B b; C c;
    try {
        B& rB1 = b;
        A& rA2 = b;
        cout << "Up-cast: Valid" << endl;

        A& rA3 = b;
        B& rB4 = dynamic_cast<B&>(rA3);
        cout << "Down-cast: Valid" << endl;

        try {
            A& rA5 = a;
            B& rB6 = dynamic_cast<B&>(rA5);
            } catch (bad_cast e) { cout << "Down-cast: Invalid: " << e.what() << endl; }

            try {
                A& rA7 = (A&)c;
                C& rC8 = dynamic_cast<C&>(rA7);
                } catch (bad_cast e) { cout << "Unrelated-cast: Invalid: " << e.what() << endl; }
            } catch (bad_cast e) { cout << "Bad-cast: " << e.what() << endl; }
    }
    return 0;
}
```

The slide also includes a sidebar with navigation links: Module 34, Partha Pratim Das, Objectives & Outline, Cast Operators, dynamic_cast, typeid Operator, Summary, and a photo of the speaker.

Now what happened when we try to do this in the earlier case when we tried to do something similar here then the resultant was that the pointer turned out to be null. Now if I am doing dealing with references of objects then certainly null reference is not possible in C++. So, this in this case you cannot have; this is an invalid case, because you cannot have a reference because that object is not of the B type it is of the generalized A type. So, this conversion reference cannot be done.

What it does? It is what is known as it throws a bad cast exception, now we have not yet discussed about exception. So, it is one way of saying that is something has gone wrong. So, when we do exceptions next you will understand how what exactly is happening, but all that it is doing it is same that something has gone wrong and we cannot proceed further. And it comes out and prints this separate message which you can see here. Similarly, here I have tried to do a casting of the reference based on the unrelated class C which also should be invalid. So, it gives you unrelated class here. So, this is the way to use the dynamic caste operator.

(Refer Slide Time: 19:29)

The screenshot shows a presentation slide titled "typeid Operator". The slide content is as follows:

- `typeid` operator is used where the **dynamic type** of a **polymorphic object** must be known and for static type identification *[Jan My - 71]*
- `typeid` operator can be applied on a type or an expression
- `typeid` operator returns `const std::type_info`. The major members are:
 - `operator==`, `operator!=`: checks whether the objects refer to the same type
 - `name`: implementation-defined name of the type *[A by PA, HE]*
- `typeid` operator works for polymorphic type only (as it uses RTTI – virtual function table)
- If the polymorphic object is bad, the `typeid` throws `bad_typeid` exception

The slide also includes a sidebar with navigation links: Module 34, Partha Pratim Das, Objectives & Outline, Cast Operators dynamic_cast, typeid Operator, and Summary. There is a small video thumbnail of the speaker on the left.

Along with the dynamic cast operator another operator which is available is known as a typeid operator which simply tries to find out that. For example, if you have a class A or something I mean whatever that class A is and you have a variable of type A or rather you have a pointer of this type pA. And of pA could be pointing to something, you do not know, what it is pointing to; it could be pointing to anything.

Now the question is, is it possible that from this pointer value I can say what is the type of pointed object, what is the type of the object that is pointing to. It would be great if we can say this is dynamic type, if we can say what is the type that, because it is a pointers which can be of own static type, but can point to number of different types.

So, typeid operator is tries to find out the dynamic type of a polymorphic object, which can exist in different forms. It should be able to compare the dynamic type of two such objects and say if there are of the same type, there of the different types. So, this is useful in that way and it can again be used only with polymorphic types not with non-polymorphic types that are you cannot use it on a non-polymorphic hierarchy or things like that.

(Refer Slide Time: 21:13)

The slide title is "Using typeid Operator: Polymorphic Hierarchy". The left sidebar contains a navigation menu with "Module 34", "Partha Pratim Das", "Objectives & Outline", "Cast Operators dynamic_cast", "typeid Operator", and "Summary". A circular video player window shows a man speaking. The main content area displays the following C++ code:

```
#include <iostream>
#include <typeinfo>
using namespace std;

// Polymorphic Hierarchy
class A { public: virtual ~A() {} };
class B : public A {};

int main() {
    A a;
    cout << typeid(a).name() << ": " << typeid(&a).name() << endl; // Static
    A *p = &a;
    cout << typeid(p).name() << ": " << typeid(*p).name() << endl; // Dynamic

    B b;
    cout << typeid(b).name() << ": " << typeid(&b).name() << endl; // Static
    p = &b;
    cout << typeid(p).name() << ": " << typeid(*p).name() << endl; // Dynamic

    A kr1 = a; A kr2 = b;
    cout << typeid(kr1).name() << ": " << typeid(kr2).name() << endl;

    return 0;
}

class A: class A *
class A *: class A
class B: class B *
class A *: class B
class A: class B
```

Annotations include red arrows pointing to the typeid operator and its arguments in the code, and handwritten notes like "type.info" and "B" near the typeid results.

So, it works something like this. This is simple hierarchy B is A, this is what you have. You have an A object and you apply typeid on that. If you apply typeid, it gives you a structure back it is called a type info structure. So, the typing for structure as I showed has different fields; and the most useful of that is a name field, which it gives some in some form the name of the class not the way we write it, but in some form.

(Refer Slide Time: 22:20)

The slide title is "Using typeid Operator: Polymorphic Hierarchy". The left sidebar contains a navigation menu with "Module 34", "Partha Pratim Das", "Objectives & Outline", "Cast Operators dynamic_cast", "typeid Operator", and "Summary". A circular video player window shows a man speaking. The main content area displays the same C++ code as the previous slide:

```
#include <iostream>
#include <typeinfo>
using namespace std;

// Polymorphic Hierarchy
class A { public: virtual ~A() {} };
class B : public A {};

int main() {
    A a;
    cout << typeid(a).name() << ": " << typeid(&a).name() << endl; // Static
    A *p = &a;
    cout << typeid(p).name() << ": " << typeid(*p).name() << endl; // Dynamic

    B b;
    cout << typeid(b).name() << ": " << typeid(&b).name() << endl; // Static
    p = &b;
    cout << typeid(p).name() << ": " << typeid(*p).name() << endl; // Dynamic

    A kr1 = a; A kr2 = b;
    cout << typeid(kr1).name() << ": " << typeid(kr2).name() << endl;

    return 0;
}

class A: class A *
class A *: class A
class B: class B *
class A *: class B
class A: class B
```

Annotations include red arrows pointing to the typeid operator and its arguments in the code, and handwritten notes like "type.info" and "B" near the typeid results, similar to the previous slide.

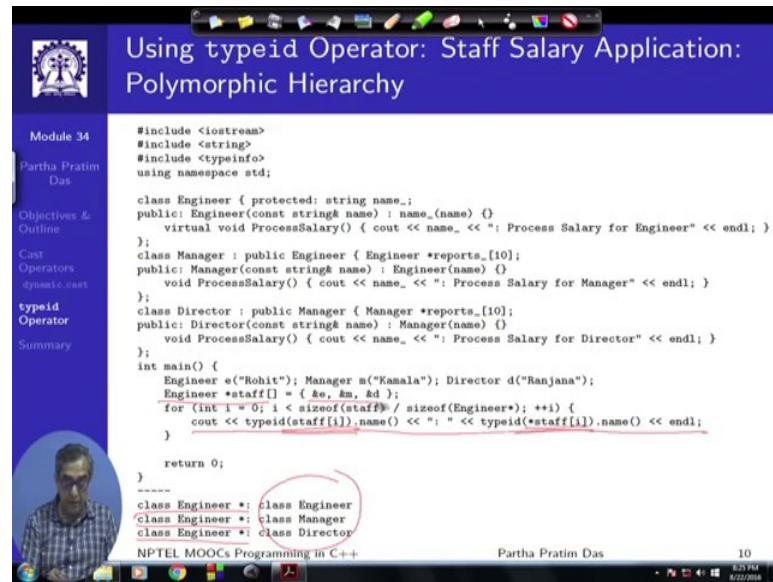
So, typeid(a).name will give the name of the type. Similarly, if I do typeid(& a).name it will give the name of that type. So, if I do this it prints class A, for this class A, it prints class A*, so that is basically the typeid output. If I take this pointer to this address, and print the pointer or the pointed object, I get A*, I get class A. Simple not a problem nothing interesting as such, it is just taking the object type and the pointer type; pointer type and the object type. Think about you have an object B, you do the same thing here print the typeid on the object and typeid on its address, you get class B and class B *.

Interesting thing starts happening here, you assign the address of B to p. p is of type A. Again in this assignment itself, I have an up-cast because p is of type A, and this is a B type address. you have up-cast. So, you are holding the B object using A type pointer. Now try to do typeid p, typeid p is static type which is A *, because it is A type. typeid star p, what is a type of star p , what the type of object it is pointing to that is class p, it is a dynamic type. So, typeid is a dynamic type operator. it tells you what is the dynamic type of that. So, earlier you did this while p was pointing to an A object, you got class A. Now it is pointing to a B type object you do here you get class B.

You can do the same thing with reference, you have a reference to A, and you have another reference of type A to object B. You just look at this, this is one reference of type A to object A. Another reference of type A, to object b if you print r1, it is class A; if you print the typeid of r2, it is class B. Because the reference is, but the reference is of type a, but what does it print typeid what does it give, it gives you the typeid of the object it is referring to which is type B. So, that is basic typeid operator.

In some cases, you could use this operator you can actually compare. caution is do not take this strings very strictly, because compilers do not guarantee that my compiler is giving telling this class A, your compiler might write it in a very different form, it might be you write it in a different case or in some other notation. So, do not take these names very strictly, what you should do is rather use the equality and inequality operator that the type info class offers, so that you can check if two objects are of the same type or they are of different types.

(Refer Slide Time: 25:23)



Module 34
Partha Pratim Das
Objectives & Outline
Cast Operators dynamic.cast
typeid Operator Summary

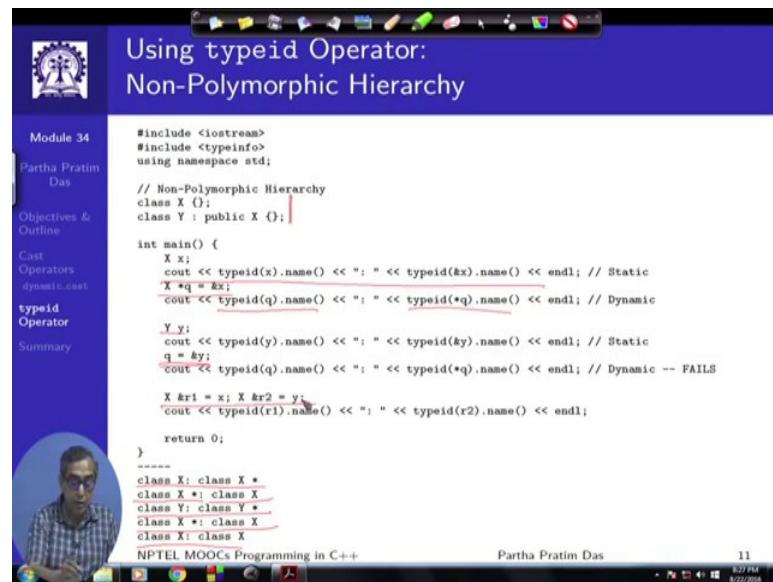
```
#include <iostream>
#include <string>
#include <typeinfo>
using namespace std;

class Engineer { protected: string name_;  
public: Engineer(const string name) : name_(name) {}  
virtual void ProcessSalary() { cout << name_ << ": Process Salary for Engineer" << endl; }  
};  
class Manager : public Engineer { Engineer *reports_[10];  
public: Manager(const string& name) : Engineer(name) {}  
void ProcessSalary() { cout << name_ << ": Process Salary for Manager" << endl; }  
};  
class Director : public Manager { Manager *reports_[10];  
public: Director(const string& name) : Manager(name) {}  
void ProcessSalary() { cout << name_ << ": Process Salary for Director" << endl; }  
};  
int main() {  
    Engineer e("Rohit"); Manager m("Kamala"); Director d("Ranjana");  
    Engineer *staff[] = { &e, &m, &d };  
    for (int i = 0; i < sizeof(staff) / sizeof(Engineer); ++i) {  
        cout << typeid(staff[i]).name() << ": " << typeid(*staff[i]).name() << endl;  
    }  
  
    return 0;  
}----  
class Engineer *; class Engineer  
class Engineer *; class Manager  
class Engineer *; class Director
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 10 8:23 PM 6/22/2016

So, this is for your workout, this is our staff salary application in the polymorphic hierarchy. I have just tried to print the typeids of the pointer type and pointed object type. So, you can say the pointers are always engineer star because our original array was of engineer star, but pointed objects are of different types as we are pointing to. You can read through this again, and understand it better.

(Refer Slide Time: 25:57)



Module 34
Partha Pratim Das
Objectives & Outline
Cast Operators dynamic.cast
typeid Operator Summary

```
#include <iostream>
#include <typeinfo>
using namespace std;

// Non-Polymorphic Hierarchy
class X {};
class Y : public X {};

int main() {
    X x;
    cout << typeid(x).name() << ": " << typeid(&x).name() << endl; // Static
    X *q = &x;
    cout << typeid(q).name() << ": " << typeid(*q).name() << endl; // Dynamic

    Y y;
    cout << typeid(y).name() << ": " << typeid(&y).name() << endl; // Static
    q = &y;
    cout << typeid(q).name() << ": " << typeid(*q).name() << endl; // Dynamic -- FAILS

    X kr1 = x; X kr2 = y;
    cout << typeid(kr1).name() << ": " << typeid(kr2).name() << endl;

    return 0;
}----  
class X: class X *  
class X *; class X  
class Y: class Y *  
class X *; class X  
class X: class X
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 11 8:27 PM 6/22/2016

For a change, if you look into a case, where we have non-polymorphic types, again I have a hierarchy, but the only difference is non-polymorphic in the sense that there is no virtual function in this. So, what is the consequence? the consequence is if the no virtual functions then the objects do not have a virtual function pointer to the table. So, they are now it is now not possible at the run time to say, what is the class from where that object actually came. So, we can demonstrate that here. So, we see x object and it is address, class x: class x*.

Again, through a pointer we set a pointer class x, another object y class y, y *. Now, you say q on this class x, x star. So, basically now you do not you cannot track that you have actually a different dynamic object residing there because all that you get is static information alone. So, similar thing will get if you do the references as well. So, if the hierarchy is non-polymorphic, then the typeid mechanism will simply not work.

(Refer Slide Time: 27:26)

The screenshot shows a presentation slide with the title "Using typeid Operator: bad_typeid Exception". The slide content includes a sidebar with navigation links like "Module 34", "Partha Pratim Das", "Objectives & Outline", "Cast Operators", "dynamic_cast", "typeid Operator", and "Summary". The main area displays C++ code and its output. The code defines a class A with a virtual constructor and a class B derived from it. It then attempts to typeid a deleted constructor of class A and a deleted copy assignment operator of class A. The output shows the typeid operator failing to provide valid information for deleted functions.

```
#include <iostream>
#include <typeinfo>
using namespace std;

class A { public: virtual ~A() {} };
class B : public A {};

int main() {
    A *pA = new A;
    try {
        cout << typeid(pA).name() << endl;
        cout << typeid(*pA).name() << endl;
    } catch (const bad_typeid& e) { cout << "caught " << e.what() << endl; }

    delete pA;
    try {
        cout << typeid(pA).name() << endl;
        cout << typeid(*pA).name() << endl;
    } catch (const bad_typeid& e) { cout << "caught " << e.what() << endl; }

    pA = 0;
    try {
        cout << typeid(pA).name() << endl;
        cout << typeid(*pA).name() << endl;
    } catch (const bad_typeid& e) { cout << "caught " << e.what() << endl; }

    return 0;
}
```

Output:

```
class A
class A *
caught Access violation - no RTTI data!
class A *
caught Attempted a typeid of NULL pointer!
```

There are some more cases where you could try this, for example, again this is a polymorphic hierarchy; I am just I will just illustrate something interesting. So, I have created a new object of the base type. I have done the typeid's, these are the two outputs, and it is fine, nothing new. I have deleted this object through pA, so it does not exist. I try to typeid again or in the pointer, I get A* which is valid because this is static type. I

tried to do typeid of *pA; I get a message caught access violation no RTTI data.

See normally, what you will expect is that, if we try to do this is the system will crash, but here that you will not do that. It will give you certain bad typeid exception. It will tell you that you are trying to find typeid for something which cannot be found, and on that bad typeid there is some message. So, if you can actually use this on a polymorphic hierarchy, you can actually used to find out if a pointer is dangling or it is pointing to a valid object. So, this gives you lot of other additional advantages also

Last I just make this pointer null. I just made p a as 0; assigned it to 0. Again I tried to find out the type this is the static type which is fine. I tried to do *pA, I got an exception attempted a typeid of null pointer. So, the typeid computation on a polymorphic hierarchy will always tell you about what is the actual dynamic type, what is a actual dynamic status of the object that is being pointed to.

Of course, I mean as a practice I would not advice that you use the typeid operator very frequently in terms of your program, for the simple reason that since it uses the virtual function table and all that information in next to make use of what is known as RTTI - run time type information, which needs to regularly track what are where the virtual function tables are and what their addresses and so on.

So, usually it is quite a lot of time taking to compute this at the run time. So, it is much better to be able to manage with just the virtual functions themselves which are pretty efficient than to explicitly use dynamic cast or certainly the typeid operator. But you do have them in your armory, so that as and when required if you are in the extreme case you could use them and make a good benefit out of that.

(Refer Slide Time: 30:46)

Module Summary

- Understood casting at run-time
- Studied dynamic_cast with examples
- Understood RTTI and typeid operator

NPTEL MOOCs Programming in C++ Partha Pratim Das 13

To summarize, we have understood casting at run time in this module. And we have started studied dynamic cast as a last type of cast operator and associated with it. We have seen that in the presence of RTTI, the typeid operator can actually find out the dynamic type of a pointer an object and if that is on a polymorphic hierarchy. So, with this module, we conclude our discussions on type casting and cast operators in C++. And we have shown that they are four operators of const cast, static cast, reinterpret cast and dynamic cast are sufficient to solve any of your cast requirements.

And with that you should minimize on; actually I would advice that you completely remove any use of C style casting in your code, just rely on these cast operators. Primarily, rely on the static cast and const cast operator, and dynamic cast when you are on a polymorphic hierarchy.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 50
Multiple Inheritance

Welcome to module 35 of Programming in C++. In this module, we will talk about Multiple Inheritance in C++. We have already discussed the inheritance mechanism in C++ at length; we have also discussed dynamic binding or polymorphism in the context of polymorphic hierarchies.

(Refer Slide Time: 00:53)

So, we will here try to specifically take a look into the multiple inheritance aspect of C++.

(Refer Slide Time: 01:01)

This slide shows the 'Module Outline' for Module 35. It features a sidebar with a logo, the title 'Module Outline', and a list of topics: 'Multiple Inheritance in C++', 'Semantics', 'Data Members', 'Overriding and Overloading', 'Access Control', 'Constructors & Destructors', 'Object Lifetime', and 'Diamond Problem'. A circular profile picture of Partha Pratim Das is also present. The main content area lists the following topics:

- Multiple Inheritance in C++
 - Semantics
 - Data Members and Object Layout
 - Member Functions
 - `protected` Access
 - Constructor & Destructor
 - Object Lifetime
- Diamond Problem
 - Exercise
- Design Choice

At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

This is somewhat advanced topic. So we would cover the multiple inheritance at a top level, and we will live some of the final issues as exercised to you all. This is the Module Outline and it will be available on the left of every slide that you see.

(Refer Slide Time: 01:29)

This slide illustrates multiple inheritance in C++. It shows a class hierarchy with three classes: `Student`, `Teacher`, and `TA`. The `TA` class inherits from both `Student` and `Teacher`. The diagram shows arrows pointing from `Student` and `Teacher` to `TA`. Below the diagram, the code defines `Student` and `Teacher` as base classes, and `TA` as a derived class that inherits from both.

```
class Student;
class Teacher;
class TA: public Student, public Teacher; // Derived Class = TA
```

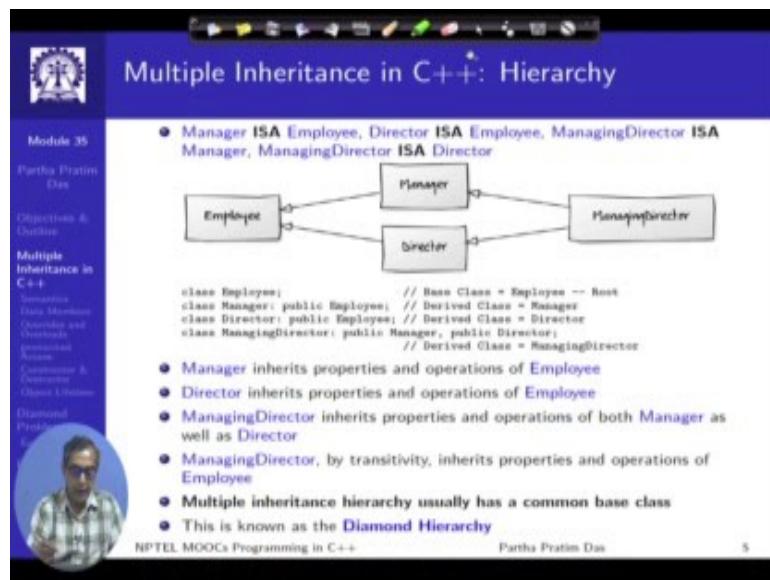
The slide also includes the following text:

- TA ISA Student; TA ISA Teacher
- TA inherits properties and operations of both Student as well as Teacher

At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

Multiple inheritance is a specific form of inheritance where a particular class has two or more base classes. We have seen the representation of inheritance, so TA is a student we know how to represent that, if we see this, I am sorry, if we look at this part then we know it represents TA is a student. Similarly TA is a teacher, so that represents it here. This is a scenario where a TA or Teaching Assistant is a student, she attends courses and tries to complete some degree, at the same time she helps some course in terms of tutorials assistance and assignment evaluation and so on. So she also performs a number of operations that are usual for the teachers. So given this we will say that the student and teacher both are base classes of the TA. And when that happens we will say that we have a situation of multiple inheritance.

(Refer Slide Time: 02:47)



So, here we show another example of multiple inheritance. There is an employee class which is kind of the root class where all employees of an organization belong, so we can say that manager ISA employee. This represents manager ISA employee. We also can say that director ISA employee. So, manager manages, director directs the policies of the company. Then we have a managing director, who is a manager as well as a director. So, there is a situation of multiple inheritance that will happen here. When multiple inheritance happens then two things would happen one is every individual inheritance involved in that multiple inheritance will inherit everything in using the rules that we

have learnt for inheritance, but in addition there could be some complications which we will have to discuss.

Then usually if we have a multiple inheritance then it is common that we have some base class which is common between the different classes which worked as a base for the multiple inheritance. So, here we see some kind of a diamond structure being created where we have the final leaf level class of the derived class here, we have intermediate base classes here, which the derived class of the leaf class actually inherits. In turn, these intermediate classes specialize from some combine concept. Because certainly if we are inheriting in a multiple way then this managing director certainly is multiply inheriting from managing and director because it has some commonality with both. So it is expected that manager and director themselves will have certain common properties and certain common operations that are here represented in terms of employee.

In terms of definition of multiple inheritance or multiple inheritance in C++ it is not mandatory to have a common base class, but it is typical that we will often have a common base class representing the whole situation.

(Refer Slide Time: 05:17)

Module 35
Partha Pratim Das
Objectives & Outline
Multiple Inheritance in C++
Semantics
Data Members
Derivations and
Inheritance generated Access
Construction &
Destruction Object Lifetime
Diamond Problem
etc

Multiple Inheritance in C++: Semantics

- Derived ISA Base1, Derived ISA Base2

```
class Base1; // Base Class = Base1
class Base2; // Base Class = Base2
class Derived: public Base1, public Base2; // Derived Class = Derived
```

- Use keyword **public** after class name to denote inheritance
- Name of the Base class follow the keyword
- There may be more than two base classes
- public** and **private** inheritance may be mixed

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, we will go into the actual syntactic details. We say generically derived is a base1, derived is a base 2, this is a generic scenario. And when we have that this is a class base 1, this is a class base2, the two base classes. And we write the derived class or the multiply inherited specialized classes; public base1, public base2. Earlier, when we had single inheritance if there is a single inheritance you just stop here, here we continue we use a comma and continue to write the next base class which is also inherited by derived. It is not that though I am showing examples where there are 2 base classes, but it is not limited to 2 base classes I can have two or more or any number of base classes in a multiple inheritance scenario.

And also as we know that the basic inheritance mechanism in C++ that is the inheritance mechanism what I mean is, ISA relationship is represented in terms of public inheritance and we have heard long discussions regarding what does that mean. But at the same time we know that there are other forms of inheritance in C++ particularly something which is known as a private inheritance which changes the visibility of the base class members in the derived class by restricting them to the private access alone. We saw that this is gives is gives us something like a semantics of IS implemented as kind of a semantics where we try to represent that if we are inheriting in private from a base class then all that we are saying that this base class actually implements the derived class.

So while we do multiple inheritance it is not necessary that these will have to be public, these could be mixed if you in fact all of them could be private in which case it will mean that both base classes are implementing certain parts of the derived class, it could also be that this is public and this is private so if that be that the case then it will mean that a derived is basically specializing from base one in the sense of ISA relationship. Whereas, the class base two implements or helps in implementing the class derived, this is a basic mechanism that we have.

(Refer Slide Time: 07:55)

Module 35
Partha Pratim Das
Objectives & Outline
Multiple Inheritance in C++
Semantics
Data Members
Overriding and Overloading Member Functions
Constructors & Destructors
Object Lifetime
Diamond Problem
etc.

Partha Pratim Das

Multiple Inheritance in C++: Semantics

- Derived **ISA** Base1, Base2
- Data Members
 - Derived class *inherits* **all** data members of all **Base** classes
 - Derived class may **add** data members of its own
- Member Functions
 - Derived class *inherits* **all** member functions of all **Base** classes
 - Derived class may *override* a member function of **any** **Base** class by *redefining* it with the **same signature**
 - Derived class may *overload* a member function of **any** **Base** class by *redefining* it with the **same name**, but **different signature**
- Access Specification
 - Derived class **cannot access private** members of **any** **Base** class
 - Derived class **can access protected** members of **any** **Base** class
- Construction-Destruction
 - A *constructor* of the **Derived** class **must first** call **all constructors** of the **Base** classes to construct the **Base** class instances of the **Derived** class – **Base** class *constructors* are called in *using order*
 - The *destructor* of the **Derived** class **must** call the *destructors* of the **Base** classes to destroy the **Base** class instances of the **Derived** class

NPTEL MOOCs Programming in C++
Partha Pratim Das

So, we will try look at the semantics. I would suggest that you compare this slide with the semantics of inheritance that we had done earlier. All those semantics are maintained so that as you move from single inheritance to multiple inheritance the basic properties remain same. So, the derived class now inherits all data members of all base classes. It is not just one base class, it inherits all data members of all base classes and it may add new data members. It inherits all member functions of all base classes again it can override or any member function of any base class it can overload any member function of any base class and so on. All these were earlier, this context of multiple base classes were not there so the semantics did not have that, but now since there are multiple base classes all of these will be possible. So, inheritance will mean that all properties and operations of each one of the base class will be inherited and they can be suitably overloaded or overridden.

Access specification will similarly have same semantics of being private, private being completely restricted to the base class. So, the derived class will not be able to access the private data members of any of the base classes. If I use protected for some data members of a base class then those data members would be available in the derived class. In terms of construction and destruction, we will see in terms of constructor. All base class objects will have to be constructed, because all base class objects will become

part of the derived class object. Now, we have two or more base classes that the derived classes is deriving from. So we also need to understand the order in which the constructors of these base classes will get executed.

So that will be in terms of listing order as we will see and when it will come to the order of destruction then the same principle that we had seen earlier that the first that is the derived class is destructed, then the base class is destructed, since there are multiple base class objects. So they will be destructed in the reversed order in which they were originally constructed. This is a summary of the semantics for multiple inheritance. There is some more details into that which will come through as we go through the examples.

(Refer Slide Time: 10:30)

The slide has a blue header with the title 'Multiple Inheritance in C++: Data Members and Object Layout'. On the left, there's a sidebar with a logo, the text 'Module 39', 'Partha Pratim Das', 'Objectives & Outline', and a list of topics including 'Multiple Inheritance in C++', 'Summary', 'Data Members', 'Object Layout', 'Access Specifiers & Constructors', 'Object Layout', and 'Diamond Problem for Multiple Inheritance'. The main content area contains the following bullet points:

- Derived **ISA** Base1, Base2
- Data Members
 - Derived class *inherits all* data members of all **Base** classes
 - Derived class may *add* data members of its own
- Object Layout
 - Derived class *layout* contains instances of *each* Base class
 - Further, Derived class *layout* will have data members of its own
 - C++ does not guarantee the *relative position* of the Base class instances and Derived class members

We start with the specific semantics of data members. So it inherits all data members of all base classes may add new data members. Now in terms of the layout therefore, we have discussed about the layout that, in the layout if a derive class inherits from a base class then it contains an instance of the base class objects. Now, since there are multiple base classes so it will have one instance each for each of the base classes. Again, like in the case of some single inheritance the C++ does not guarantee the relative position of the base class instances. How they will be organized whether first the base class objects

will be there then the derived class object members will be there and so on, that specific order is not decided not given by the standard.

(Refer Slide Time: 11:28)

The slide title is "Multiple Inheritance in C++: Data Members and Object Layout". It features a logo of a person in a circular emblem at the top left. On the left, there's a sidebar with "Module 35", "Partha Pratim Das", "Objectives & Outline", "Multiple Inheritance in C++", and "Data Members". A small circular video player shows a person speaking. The main content area contains C++ code:

```
class Base1 { protected:
    int i;
    int data_1;
public: // ...
};

class Base2 { protected:
    int j;
    int data_2;
public: // ...
};

class Derived : public Base1, public Base2 {
    int k;
public: // ...
};
```

Below the code is a diagram titled "Object Layout" showing three objects: "Object Base1", "Object Base2", and "Object Derived". "Object Base1" has a box labeled "i" and "data_1". "Object Base2" has a box labeled "j" and "data_2". "Object Derived" has three stacked boxes labeled "i", "data_1", "j", "data_2", and "k". A note at the bottom says "Object Derived has two data_1 member! Ambiguity to be resolved with base class name: Base1::data_1 & Base2::data_1." The footer includes "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and a page number "9".

If we look into the instance you have a base class b1 with two members i and data. I have another base class b2 with two members; j and data. And I have a derived class which derives from base 1 as well as base 2 and it adds a member k. If I look at one object of base 1 type it will have a member i and a member data, if I look at the object of derived base class b base 2 then it have an instance like this it will have a member j and data. So when we construct a derived class object will have an instance of the base 1 object which one class will have instance of the base 2 class and will have whatever data members we have added in the derived. So, you can clearly see that it is just a direct extension of the semantics of data members that we had or the semantics of layout that we had in case of the single inheritance.

Of course, as I said there could be some pitfalls. For example, as you can see here that base1 has declared a member data and base2 also has a member by the exactly the same name. Since base classes are independent of the derived class you cannot control that they would not have members with the same name. When they have members with same name so the object derived this object has two data members by the same name. If I say

that my object is derived d then I want to write derived data which I am authorized to write because data is protected here as well as it is protected here so derived class has access to them.

But if I try to write this certainly the compiler will say that I am confused, because there are two members by the same name. This is a case of ambiguity so if two base classes two more base classes have data members by the same name then the responsibility would lie with the programmer or the designer of the derived class to resolve that ambiguity. You will have to refer to the members explicitly with the class name qualifier. This will not be acceptable by the compiler, but I can write d. base1 :: data, if I write this it will mean this data member or it will mean this data member in the object if this is d. But if I write d.base2 :: data, then it will mean that this data member so that resolution additionally would required to be done. This is one added complexity that will have in terms of the multiple inheritance.

(Refer Slide Time: 14:39)

Multiple Inheritance in C++:
Member Functions – Overrides and Overloads

- Derived **ISA** Base1, Base2
- Member Functions
 - Derived class *inherits all* member functions of all **Base** classes
 - Derived class may *override* a member function of **any** Base class by *redefining* it with the **same signature**
 - Derived class may *overload* a member function of **any** Base class by *redefining* it with the **same name; but different signature**
- Static Member Functions
 - Derived class *does not inherit* the static member functions of **any** Base class
- Friend Functions
 - Derived class *does not inherit* the friend functions of **any** Base class

Now, let us move on to the Member Functions - Overrides and Overloads. As I already said the all member functions are inherited from all base classes and you can override any member function, overload a member function from a base class. And like in single

inheritance the static member functions and the friend functions are not inherited by the base class here as well.

(Refer Slide Time: 15:09)

The slide has a blue header with the title 'Multiple Inheritance in C++: Member Functions – Overrides and Overloads'. Below the title is a portrait of a man with glasses and a plaid shirt. On the left, there's a sidebar with navigation links: 'Module 35', 'Partha Pratim Das', 'Objectives & Outline', 'Multiple Inheritance in C++', 'Solutions', 'User Manual', 'Overrides and Overloads', 'Previous Session', 'Contentions & Questions', 'Object Lifetime', and 'Diamond Problem'. The main content area contains the following C++ code:

```

class Base1 { protected:
    int i_1;
    int data_1;
public: Base1(int a, int b) : i_(a), data_(b);
    void f(int) { cout << "Base1::f(int)" << endl;
    void g() { cout << "Base1::g()" << endl;
};

class Base2 { protected:
    int i_2;
    int data_2;
public: Base2(int a, int b) : i_(a), data_(b);
    void h(int) { cout << "Base2::h(int)" << endl;
};

class Derived : public Base1, public Base2 {
    int k_;
public: Derived(int x, int y, int u, int v, int w) {
    void f(int) { cout << "Derived::f(int)" << endl; // -- Overrides Base1::f(int)
    // -- Inherited Base1::g()
    void h(string) { cout << "Derived::h(string)" << endl; // -- Overloaded Base2:: h(int)
    void u(char) { cout << "Derived::u(char)" << endl; // -- Added Derived::u(char)
};

Derived d(1, 2, 3, 4, 5);

d.f(8); // Derived::f(int) -- Overrides Base1::f(int)
d.g(); // Base1::g() -- Inherited Base1::g()
d.h("pppl"); // Derived::h(string) -- Overloaded Base2:: h(int)
d.u('a'); // Derived::u(char) -- Added Derived::u(char)

```

At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

If we look at an example, just look at this carefully the same set of base classes; base class base 1, class base 2, and class derived which specializes from base 1 as well as base 2. Here, I have two member functions f and g, and here in base 2 I have a member function h. And what I have done is this is f int in the derived class I have included a member function with the same signature which means this is a case of overriding. So, the derived class is overriding the f member function from base1. Base1 also has a member function g and derived class has no mention of any member function by the name g, therefore it inherits g simply and would be able to use that. And when it uses g it will mean the g member function of the base1 of base class. Base class 2 has a member function h and the derived class introduces a member function h with the different signature. We know what will be the effect, this h new h derived :: h will hide base2 :: h which was taking integer and now you will be able to call h for a derived class object with a string parameter. So this is a case of overloading. This is simple that derived class can have a new member function e added to the kitty and that can be used.

In this context, if we have a derived class object c if I do c dot f 1 then f function in the derived class will be called because base1 ::f have been overridden. If we call c dot g it is a member function of base 1 will be called because it has been inherited, if we called c dot h with, this is where I have used a constant char star kind of parameter which will get cast automatically implicitly to string. So overloaded h function in derived will be called not the base class function because that has got hidden, and if I call c dot e with character a then it will call the e member function that has been introduced in the derived class.

This is the basic story of overriding and overloading that will happen, and certainly like the data member it must be occurring in your mind that what happens if 2 base classes have one function which has the same name.

(Refer Slide Time: 18:06)

Ambiguous Calls	Unambiguous Calls
<pre>class Base1 { public: Base1(int a, int b) : i_(a), data_(b) { void f(int) { cout << "Base1::f(int)" << endl; void g() { cout << "Base1::g()" << endl; } }; class Base2 { public: Base2(int a, int b) : j_(a), data_(b) { void f(int) { cout << "Base2::f(int)" << endl; void g(int) { cout << "Base2::g(int)" << endl; } }; class Derived : public Base1, public Base2 { public: Derived(int x, int y, int u, int v, int w); using Base1::f; using Base2::g; }; Derived d(1, 2, 3, 4, 5); d.f(6); // Base1::f(int) or Base2::f(int)? d.g(); // Base1::g() or Base2::g()? d.f(0); // Base1::f(int) or Base2::f(int)? d.g(); // Base1::g() or Base2::g()?</pre>	<pre>class Base1 { public: Base1(int a, int b) : i_(a), data_(b) { void f(int) { cout << "Base1::f(int)" << endl; void g() { cout << "Base1::g()" << endl; } }; class Base2 { public: Base2(int a, int b) : j_(a), data_(b) { void f(int) { cout << "Base2::f(int)" << endl; void g(int) { cout << "Base2::g(int)" << endl; } }; class Derived : public Base1, public Base2 { public: Derived(int x, int y, int u, int v, int w); using Base1::f; // Hides Base2::f using Base2::g; // Hides Base1::g; }; Derived d(1, 2, 3, 4, 5); d.f(6); // Base1::f(int) d.g(); // Base2::g(int) d.Base1::f(0); // Base2::f(int) d.Base1::g(); // Base1::g()</pre>
<ul style="list-style-type: none"> • Overload resolution does not work between Base1::g(int) and Base2::g() • using hides other candidates • Explicit use of base class name can resolve (weak solution) 	

What happens if 2 base classes have a common name for a member function. So I illustrate that in this slide, again I have base 1, base 2, the derived class derives from the same base classes. The difference that is being made is, I have f here and I have f here both in base 1 and base 2 and their signatures are same. I have g here in base 1, I have g here in base 2 their signatures are different. Then in further moment ignore this part. Then in the derived class let us say you do not have member function at all just ignore this part, this you should not consider right now. Now, I try to write c dot f the question

is what is c dot f? Is it base 1 dot f; base1 :: f or it is base 2 colon colon f you have no way of knowing because it has got two versions of f.

What is c dot g? I have passed a parameter 5 expecting that the compiler would be able to resolve that it is base 1 :: g because, I am sorry there is a small type of here this should not be int, this should be int. If I call c dot g 5 then it I would expect that base2 :: g will be called, but unfortunately the compiler would not be able to do that. The reason it will be not able to do that it is a fact that overload is resolved only between same name spaces. If you have two different name spaces then the compiler has no track of what the names are going to be. So, g function in base 1 and g function in base 2 are in into two different name spaces of two different classes so the overload resolution does not kick in here.

So consequently, c.g(5) or c. g() without any parameter both of them will actually turn out to be ambiguous also. In gist all of these four calls will turn out to be ambiguous and the compiler would not be able to resolve between them. So, if you want to make them resolve that ambiguity you have a simple way of saying that the basic issue that is happening is here you have as you inherit from base1 and base2 you get two copies of function f who wants to come into the derived class. Now, we have learnt about the using declaration for the parent class function, so you can use that. Suppose you say using base 1 colon colon f, if you say I am using base 1 colon colon f then what it will happen is, this function will be included in derived and this function will not be included, this function base two's function will be hidden.

Similar I can do for the g function as well, say I am using base 2 colon colon g which means this g function will be included, but base ones g function will be hidden. In this context now if I would like to call c dot f then it will call base 1 colon colon f because I have been using this. If I do c. g (5) it will call base 2 colon colon g because I am using the g function of base 2. Now in this if I also want, there is a situation that in some case I want to actually also access the f function of base 2 then like I did in case of data members I can explicitly write c dot base 2 colon colon f 3 in which case it will actually call f member function of the base two class.

Even though in the derived class I have said I am using base 1 colon colon f. What actually using does, using basically allows me to do a short cut that I do not need to qualify the name of the member function with the base class name and I can use it as a default as I am doing here, but I still always have the option of actually providing the qualified name for the member function, like I did for the data member and use the other members in that form.

So, this is what is additionally required over single inheritance which will be very common because it is quite likely that the base classes that you are inheriting from may have one or more member functions which are have the same name between themselves. As we have seen there it does not matter in terms of what actually are their signature what matters is they have the same name, and if they have the same name then the derived class cannot use them without any using qualification.

(Refer Slide Time: 23:22)

The screenshot shows a presentation slide titled "Multiple Inheritance in C++: Access Members of Base: protected Access". The slide has a dark blue header and footer. The main content area contains a bulleted list under the heading "Access Specification".

- Access Specification
 - Derived class *cannot access private* members of *any* Base class
 - Derived class *can access protected* members of *any* Base class

The sidebar on the left lists "Module 35" topics: Partha Pratim Das, Objectives & Outline, Multiple Inheritance in C++, Supporting Data Members, Overriding and Overloading, **protected Access**, Access Specifier & Derivation Order, Diamond Problem, and References. At the bottom of the sidebar is a portrait of a man.

The footer contains the text "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" along with a page number "13".

Coming to the access members of a base, the protected access will allow any derived class object to access the protected members of the base class of any of the base class there is nothing to add in terms of multiple inheritance here. So, all that we have learnt in case of single inheritance will simply apply so we will skip further discussion on this aspect.

(Refer Slide Time: 23:46)

Module 35
Partha Pratim Das
Objectives & Outline
Multiple Inheritance in C++
Inheritance Hierarchy
Overriding and Overloading
Protected Members
Constructor & Destructor
Diamond Problem
Final Class
Virtual Function
Virtual Base Class
Virtual Destructor

Multiple Inheritance in C++:
Constructor & Destructor

- Constructor-Destructor
 - Derived class *inherits all* Constructors and Destructor of Base classes (but in a different semantics)
 - Derived class *cannot override or overload* a Constructor or the Destructor of *any* Base class
- Construction-Destruction
 - A *constructor* of the Derived class *must first call all constructors* of the Base classes to construct the Base class instances of the Derived class
 - Base class *constructors* are called in *Refining order*
 - The *destructor* of the Derived class *must call the destructors* of the Base classes to destruct the Base class instances of the Derived class

NPTEL MOOCs Programming in C++ Partha Pratim Das 14

Let me move onto the Constructor, Destructor. The constructor, destructors of the derived class will inherit all the constructor destructors of the base classes, but in a different semantics as we saw in case of the single inheritance because it cannot directly inherit that because it has a different name, it adds name of the base class and certainly you cannot override or overload the constructor, destructor in any way. So, if we see with that then we can see that there are base class has a constructor here.

(Refer Slide Time: 24:22)

Multiple Inheritance in C++:
Constructor & Destructor

```
class Base1 { protected: int i,_1, int data_1;
public: Base1(int a, int b) : i(a), data_1(b) { cout << "Base1::Base1() " ; }
        ~Base1() { cout << "Base1::~Base1() " ; }
};

class Base2 { protected: int j,_2, int data_2;
public: Base2(int a = 0, int b = 0) : j(a), data_2(b) { cout << "Base2::Base2() " ; }
        ~Base2() { cout << "Base2::~Base2() " ; }
};

class Derived : public Base1, public Base2 { int k,_3;
public: Derived(int x, int y, int z) :
        Base1(x, y), Base2(z) { cout << "Derived::Derived() " ; }
        // Base1::Base1 explicit, Base2::Base2 default
        ~Derived() { cout << "Derived::~Derived() " ; }
};

Base1 b1(2, 3);
Base2 b2(5, 7);
Derived d(5, 3, 2);

Object Layout
Object b1 Object b2 Object d
[ 2 3 ] [ 5 7 ] [ 5 3
               0 0
               2 ]
```

Partha Pratim Das

The derived class has a constructor and, sorry the second base class has another constructor and in the derived class has to; here is invoking the constructor of the base one. So what will happen? Now, it has to construct both the base class objects; the fact that is invoking base 1 means that the base 1 constructor will be invoked through this and since it is skipped base 2, the base 2 must have a default constructor which will be invoked after that. So, if I have just invoked the base 1 constructor and base 2 does not have a default constructor then I will have a compilation error, because to be able to construct a derived class object I need construct both base 1 and base 2 kind of objects.

So, if we see the instance, this is an object of base1 type; this is an object of base2 type this is what we can constructed. Here you can see the object of a derived type being constructed where the base class 1 has the instance 5 3 which got created through this. The instance of base class 2 is by default so it has 0 0 as members and this is a data member of the derived class. This is the basic dynamics of the construction process.

(Refer Slide Time: 25:48)

The slide title is "Multiple Inheritance in C++: Object Lifetime". It features a sidebar with course details: Module 35, Partha Pratim Das, Objectives & Outline, Multiple Inheritance in C++, Semantics, Data Members, Operator and Overload Resolution, General Issues, Constructors & Destructors, Object Lifetimes, and Diamond Problem. A video thumbnail of the speaker is on the left.

```
class Base1 { protected: int i_1, int data_1;
public: Base1(int a, int b) : i_1(a), data_1(b) { cout << "Base1::Base1() ";}
    ~Base1() { cout << "Base1::~Base1() ";}
};

class Base2 { protected: int j_2, int data_2;
public: Base2(int a = 0, int b = 0) : j_2(a), data_2(b) { cout << "Base2::Base2() ";}
    ~Base2() { cout << "Base2::~Base2() ";}
};

class Derived : public Base1, public Base2 { int k_3;
public:
    Derived(int x, int y, int z) {
        Base1(x, y), Base2(z) { cout << "Derived::Derived() ";}
        // Base1::Base1 explicit, Base2::Base2 default
        ~Derived() { cout << "Derived::~Derived() ";}
    }
};

Derived d(5, 3, 2);

Construction O/P: Base1::Base1(): 5, 3 // Obj. d.Base1
Base2::Base2(): 0, 0 // Obj. d.Base2
Derived::Derived(): 2 // Obj. d.Derived

Destruction O/P: Derived::~Derived(): 2 // Obj. d
Base2::~Base2(): 0, 0 // Obj. d.Base2
Base1::~Base1(): 3, 5 // Obj. d.Base1
```

A callout box contains: • First construct base class objects, then derived class object
• First destruct derived class object, then base class objects

NPTEL MOOCs Programming in C++ Partha Pratim Das 16

If you put messages into the base class constructor and destructors and so on and the derived class constructor/destructor, then you will be able to see that the first the base class 1 is constructed because it is first in the list, then the base class 2 because it is second in the list, and then the derived class constructed and the destruction happens exactly in the reverse order.

(Refer Slide Time: 26:20)

The slide title is "Multiple Inheritance in C++: Diamond Problem". It features a sidebar with course details: Module 35, Partha Pratim Das, Objectives & Outline, Multiple Inheritance in C++, Semantics, Data Members, Operator and Overload Resolution, General Issues, Constructors & Destructors, Object Lifetimes, and Diamond Problem. A video thumbnail of the speaker is on the left.

- Student ISA Person
- Teacher ISA Person
- TA ISA Student; TA ISA Teacher

Diagram showing inheritance relationships:

```
graph LR; Person --> Student; Person --> Teacher; Student --> TA;
```

Code examples:

```
class Person; // Base Class = Person -- Root
class Student: public Person; // Base / Derived Class = Student
class Teacher: public Person; // Base / Derived Class = Teacher
class TA: public Student, public Teacher; // Derived Class = TA
```

- Student inherits properties and operations of Person
- Teacher inherits properties and operations of Person
- TA inherits properties and operations of both Student as well as Teacher
- TA, by transitivity, inherits properties and operations of Person

NPTEL MOOCs Programming in C++ Partha Pratim Das 17

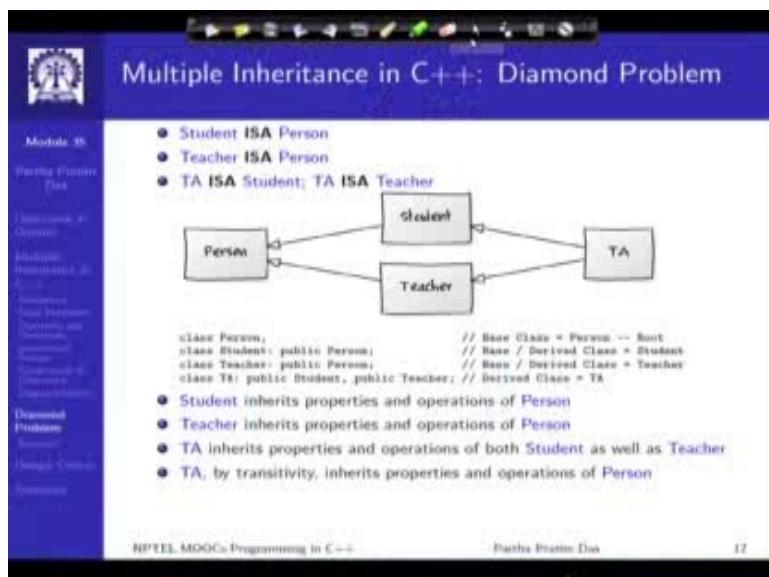
This is the basic mechanism of inheritance that goes on in terms of the multiple cases of base class being there for any particular derived class problem.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 51
Multiple Inheritance (Contd.)

Welcome back to module 35 of Programming in C++. We have been discussing multiple inheritance and we have seen the basic mechanism of construction, destruction, layout, data members, member functions and what happens, if between the multiple base classes, if the data member or the member functions have duplicate same names.

(Refer slide Time: 00:40)



Now, we will look into some of the more integrate use scenarios and let us say that we have the students teacher TA scenario, where a TA is a student, TA is a teacher and both of them are persons. So, we have a diamond kind of situation and we call this as a diamond problem we will see why we call this is as diamond problem. I have already explained that it is very common that you will have a common base class to the base classes of multiply inherited class.

(Refer slide Time: 01:13)

The slide title is "Multiple Inheritance in C++: Diamond Problem". It features a logo of a person in a graduation cap. The left sidebar contains a navigation menu with items like "Module 35", "Partha Pratim Das", "Objectives & Outcome", "Multiple Inheritance in C++", "Inheritance", "Data Members", "Overriding and Overloading", "Constructors & Destructors", "Object Lifetime", "Diamond Problem", and "Summary". A video thumbnail of the speaker is also present.

The main content area contains the following C++ code:

```
#include<iostream>
using namespace std;

class Person { // Data members of person
public: Person(int x) { cout << "Person::Person(int)" << endl; }
};

class Faculty : public Person { // data members of Faculty
public: Faculty(int x) : Person(x) { cout << "Faculty::Faculty(int)" << endl; }
};

class Student : public Person { // data members of Student
public: Student(int x) : Person(x) { cout << "Student::Student(int)" << endl; }
};

class TA : public Faculty, public Student {
public: TA(int x) : Student(x), Faculty(x) { cout << "TA::TA(int)" << endl; }
};

int main() {
    TA ta(30);
    return 0;
}
```

A red box highlights the line "Person::Person(int)". A callout bubble points to it from a handwritten note: "Two instances of base class object (Person) in a TA object!"

To the right of the code is a hand-drawn diagram. It shows a vertical stack of four boxes representing objects. From bottom to top, they are labeled "Person", "Faculty", "Student", and "TA". Arrows point from each base class ("Person", "Faculty", "Student") to their respective derived class ("TA"). A handwritten note next to the "TA" box says "TA".

At the bottom of the slide, there is footer text: "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

So, let us try to look into the code person is a class, here I have called it faculty, in the example in the code I have called it faculty, which means teacher is a class student is a class so they inherit from a person. So, and then TA inherits from faculty and student both, so this is the scenario. And just look at the every constructor has a message to see what is happening at the construction. So, in the construction happens certainly the base class will have to get constructed. So, what will have to get constructed for constructing a TA object, the faculty has to get constructed.

Now, faculty has a specialising from person. So, what will need to be done for constructing a faculty object a person has to get constructed, so these two construct a faculty object. Then the student has to get constructed and student is specialising from person. So, if I want to construct a student I will need to construct a person again. So, another person will get constructed and then student will get constructed, then TA will get constructed. So, if I look into the total object scenario, then I have two base classes this is faculty and this is a student. And this will have a person, this will have a person other than that you will have different faculty data here, you will have the student data here, you will have the derive class this is the TA object. So, you will have TA data here these are the TA data.

But this is how the construction will actually happen. So, what is interesting to note that when you construct like this our basic principle of single inheritance tell us that there will have to be two person objects in the same TA object. Because, otherwise the faculty cannot be constructed because it needs the base object to be embedded in that, student cannot be constructed because it needs the person to be embedded in that. Therefore, TA needs both faculty and student to be embodied that. So, you will have two instances of the base class object and this is certainly not a very desirable situation, because certainly TA as a person will have only one set of attributes if I have two instances of the person class and how do I going to resolve in that, how would are you going to maintain the data.

(Refer slide Time: 03:54)

**Multiple Inheritance in C++:
virtual Inheritance – virtual Base Class**

```
#include<iostream>
using namespace std;

class Person { // Data members of person
public: Person(int x) { cout << "Person::Person(int)" << endl; }
    ~Person() { cout << "Person::~Person()" << endl; } // Default ctor for virtual inheritance
};

class Faculty : virtual public Person { // data members of Faculty
public: Faculty(int x) { cout << "Faculty::Faculty(int)" << endl; }
};

class Student : virtual public Person { // data members of Student
public: Student(int x) { cout << "Student::Student(int)" << endl; }
};

class TA : public Faculty, public Student {
public: TA(int x) : Student(x), Faculty(x) { cout << "TA::TA(int)" << endl; }
};

int main() {
    TA ta(30);
    return 0;
}
```

Diamond Problem

Hand-drawn diagram illustrating the diamond problem:

```

class Person {
public: Person();
    ~Person();
};

class Faculty : virtual public Person {
public: Faculty();
};

class Student : virtual public Person {
public: Student();
};

class TA : public Faculty, public Student {
public: TA();
};

Diagram:
    +-----+
    | Person |
    +-----+
    +-----+
    | Faculty |
    +-----+
    +-----+
    | Student |
    +-----+
    +-----+
    | TA |
    +-----+
    |
    +-----+
    | Faculty |
    +-----+
    +-----+
    | Student |
    +-----+
    +-----+
    | TA |
    +-----+
  
```

Notes:

- Inference a default constructor for root base class Person
- Prefix every inheritance of Person with virtual
- Only one instance of base class object (Person) in a TA object!

So, that leads to that what is known as virtual inheritance and multiple inheritance. What it say is we use the keyword virtual, let me use a red again, use a keyword virtual before or after it does not matter as to, whether you write it after public, you can write it before public also, but you use a keyword virtual, where you are inheriting. When you do that then the inheritance become virtual which means that TA has to be constructed your first thing is faculty will have to be constructed.

Now, if you say that faculty inherits person in a virtual way then it knows that some other specialised class is being constructed for which there could be multiple base classes. So, the faculty does not construct its person class, it does not where construct is person object, does not construct that instance. Similarly, when we do student it does not construct the person instance of the student class, which otherwise I would require. But the process constructs one common person instance for both faculty as well as student.

Now, how will this get constructed, now certainly faculty is not constructing a student is not constructing is because they are virtually inheriting from person. So, this has to be a default constructor, this has to be a process of virtual inheritance that one single base class, unique base class will instance will get constructed. Therefore, I have introduced a default constructor for the person class here. Now, we can see that person class is construct instances is constructed only once and based on that the faculty and student instances. So, if we look into the base class part of the faculty you get this person, if you look into the base class part of the student you again get the same person instance, if you look into certainly the base class part of the TA then of course, you get the same person. So, you kind of unify the instance of the base class.

So, this when we use virtual inheritance and construct the hierarchy in this way to avoid having multiple instances of the route class, multiple instance of the diamond class into the derive class instance, we use virtual inheritance and such classes as known as virtual base class. So, this is a virtual base class, this is a virtual base class, these are virtual base classes VBCs. Because they will not directly construct their base class part; instance of the base class part that will be common with the total, derive class object. Now, in this process this solves a basic problem of object layout in case of diamond, that is this will happen if you have diamond. So, that this class is getting constructed here as well as getting constructed here. So, virtual inheritance basically eliminates that problem, but with one small restriction that for doing this. since we have done this automatically we could use only the default constructor the person class. So, what if I want to pass parameters to the person class that is what if I want to call this constructor.

(Refer slide Time: 07:40)

The slide title is "Multiple Inheritance in C++: virtual Inheritance with Parameterized Ctor". It features a sidebar with course navigation and a video player showing a person speaking.

```
#include<iostream>
using namespace std;
class Person {
public: Person(int x) { cout << "Person::Person(" << endl; }
~Person() { cout << "Person::Person()" << endl; }
};
class Faculty : virtual public Person {
public: Faculty(int x) :Person(x) { cout << "Faculty::Faculty(" << endl; }
~Faculty() { cout << "Faculty::Faculty()" << endl; }
};
class Student : virtual public Person {
public: Student(int x) :Person(x) { cout << "Student::Student(" << endl; }
~Student() { cout << "Student::Student()" << endl; }
};
class TA : public Faculty, public Student {
public:
    TA(int x) : Student(x), Faculty(x), Person(x) { cout << "TA::TA(" << endl; }
~TA() { cout << "TA::TA()" << endl; }
};
int main() {
    TA ta(30);
    return 0;
}
-----
```

Person::Person(int)
Faculty::Faculty(int)
Student::Student(int)
TA::TA(int)

+ Call parameterized constructor of root base class Person from constructor of TA class

NPTEL MOOCs Programming in C++ Partha Pratim Das 20

It is that is the very simple way of doing this in multiple inheritance, all that you need to do is these remains same you are virtually inheriting, this virtually inheriting this, these remain to be VBCs virtual base classes all that remain same. The only difference is in the constructor of the TA class, in the constructor of that is, in the constructor of the derive class you explicitly call the constructor of the route class and there you pass the parameters. So, what will happen if you do this then, this will be the first to get constructed which will call this constructor which has parameter.

Then according to this order this will be the next to get called and this will be the last to get called that is why you will see that it is person faculty student and TA constructor in that order that it will get constructed. And you still now have that original problem resolved you have a unique instance of the base class, but you have been able to pass parameters to the constructor. So, that is the basic way to construct a multiple inheritance hierarchy.

(Refer slide Time: 08:50)

Multiple Inheritance in C++: Ambiguity

```
#include<iostream>
using namespace std;
class Person {
public: Person(int x) { cout << "Person::Person(int)" << endl; }
    Person() { cout << "Person::Person()" << endl; }
    virtual ~Person();
    virtual void teach() = 0;
};
class Faculty : virtual public Person {
public: Faculty(int x) :Person(x) { cout << "Faculty::Faculty(int)" << endl; }
    virtual void teach();
};
class Student : virtual public Person {
public: Student(int x) :Person(x) { cout << "Student::Student(int)" << endl; }
    virtual void teach();
};
class TA : public Faculty, public Student {
public:
    TA(int x):Student(x), Faculty(x) { cout << "TA::TA(int)" << endl; }
    virtual void teach();
};

★ In the absence of TA::teach(), which of Student::teach() or Teacher::teach() should be inherited?
```

Partha Pratim Das

Module 35

Partha Pratim Das

Objectives & Outline

Multiple Inheritance in C++

Semester Data Structure and Algorithms

Downloaded from NPTEL

Diamond Problem

NPTEL MOOCs Programming in C++

Partha Pratim Das

21

And you can now go ahead and make that polymorphic and a teach member method say to the base class of person make it purely virtual, because a person you do not know how a person can teach, so instantiate that, implement that in faculty as a just a non-pure virtual function, instantiate that again in student and so on. But as you do that certainly you will have a conflict in terms of whether you will use this or you will use this and on a polymorphic hierarchy certainly you cannot specify us to which member function you will use like you did in case of a non-polymorphic using method. So, the only way you can leverage this is actually override both of them using a new member function teach in the TA class as well. If you do not do this then certainly, when you do when you try to do an instance of TA it will say that we cannot do that because you do not know which of these two teach function should this object use.

So, once you have over written this so that means, you directly hide this in the virtual function table then you can that the code will work and then depending on what you want to do. If you want to reuse this function then within the implementation of teach in the TA class you could certainly refer to it by faculty colon colon teach or if you want to reuse teach of the student class you can use student colon colon teach or you could implement it on its own. So, this is the basic issue of ambiguity in case of multiple

inheritance with diamond structure, we will not have this unless you have a diamond structure. The reason you are getting this is, because you have the diamond.

Therefore, since you have the diamond there are two ways that the teach function can reach here, the two ways that teach methods can reach here, if you have more base classes and a common diamond then you will have more ways of doing that, but two is enough for the confusion. So, here you do not know whether you should use teach of this or you should use teach of this. So, that is a basic problem and that leads to if you want to really have a generic multiple inheritance hierarchy that leads to several issues. And more often many of the organisations prescribe that you should not actually use this should not use diamond in the multiple inheritance scenarios of course, that restricts the total use of multiple inheritance in a severe way. But this is a serious interpretation problem which will have to live with.

(Refer slide Time: 11:44)

Module 35
Partha Pratim Das
Objectives & Outline
Multiple Inheritance in C++
Scenario Data Members
Overriding and Overloading
Constructors
Casting & Dynamic Object Lifetime
Diamond Problem
etc.

Multiple Inheritance in C++: Exercise

```
class A {
public:
    virtual "A() { cout << "A():A()" << endl; }
    virtual void foo() { cout << "A::foo()" << endl; }
};
class B : public virtual A {
public:
    virtual "B() { cout << "B():B()" << endl; }
    virtual void foo() { cout << "B::foo()" << endl; }
};
class C // : public virtual A {
public:
    virtual "C() { cout << "C():C()" << endl; }
    virtual void foobar() { cout << "C::foobar()" << endl; }
};
class D : public B, public C {
public:
    virtual "D() { cout << "D():D()" << endl; }
    virtual void foo() { cout << "D::foo()" << endl; }
    virtual void foobar() { cout << "D::foobar()" << endl; }
};

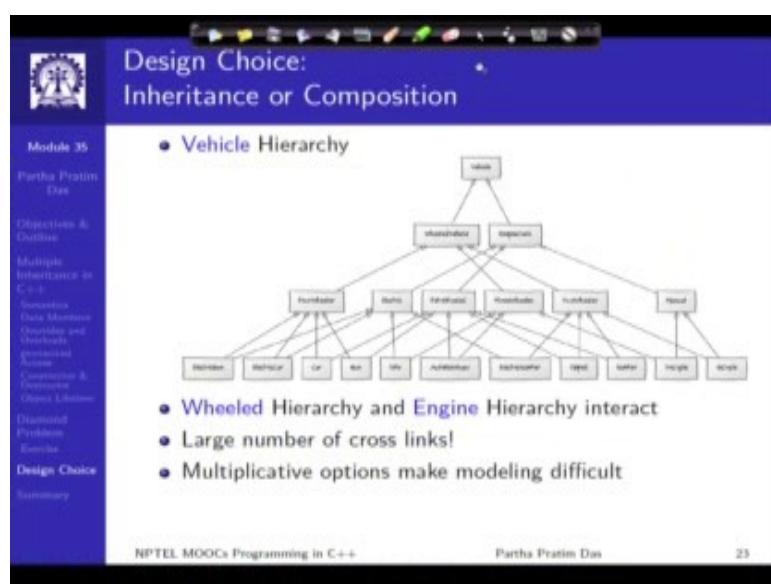
// Consider the effect of calling foo and foobar for various objects and various pointers.
```

NPTEL MOOCs Programming in C++
Partha Pratim Das
22

So, in this context, there is a; this is an exercise, where there is a class A, there is a class B. So, there is a class A, class B and then there is a class C and then there is a class D. So, this is a scenario like this. So, this is again a scenario of multiple inheritance, but this is not exactly a diamond. So, on this there are different member functions foo and foobar defined. So, I would just suggest that you having known all that you have studied in an

inheritance in casting as well as in multiple inheritance. You try to create instances of different class object, try to take pointers of different class types and try to shake out as to which how you can invoke the different member functions on this hierarchy. Of course, you can finally, make the whole thing complicated by also letting C derive from A, the moment you allow C derive from A you get into diamond and you will have lot of interesting problem of ambiguity that come in.

(Refer slide Time: 12:58)



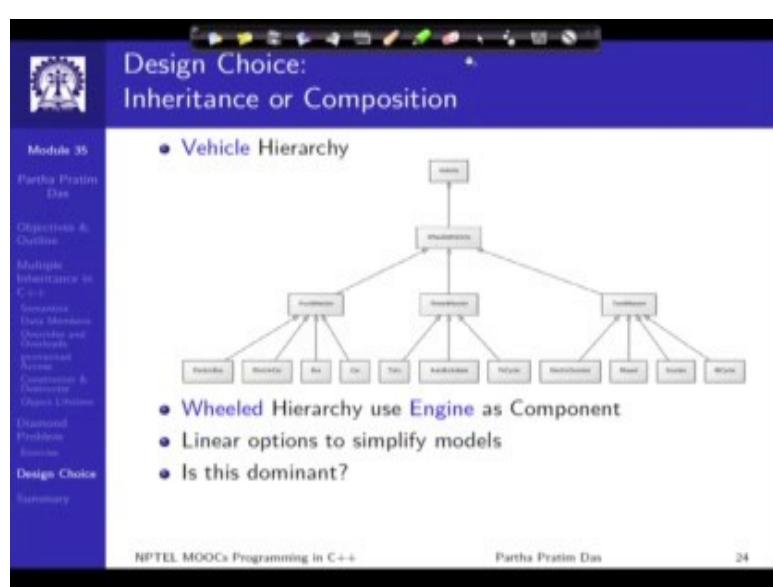
So, finally, before I close, I would like to just give you glimpse of a issue of design choice, as to whether we can always model in terms of inheritance and in place of that we could also do composition the other way hierarchy. And there is always a trade off in the designers to whether you should do inheritance or you should do composition. So, just may have created an example here to show you what kind of difficulties you accept. For example, I am looking at vehicle hierarchy and these are the primary you know sub classes of the vehicle that is you are checking out is the class of wheeled vehicles that exist in the world.

And where different types of driving mechanism, the engine class are basically the different drive mechanism that can happen for that and if you look into that then in terms of the wheeled one, you may have a two wheeler, you may have a three wheeler and you

may have a four wheeler these are the different options. And in terms of the engine class you may have a manual drive you could have a petrol fluid drive you could have a electric drive these are the all different type. So, if you have all these, so is basically you have two major hierarchies one based on the wheel drive, one based on the engine class. So, if you look at that then based on these you have a whole lot of IF classes that come in based on different combinations, like I can talk about a bicycle which is manual and two wheeler, tricycle it is manual and three wheeler. So, that is the multiple inheritance happening, I have a car which is a four wheeler and petrol fluid I have an electric car, which is four-wheeler but electric fluid and so on.

So, depending on I have three types of wheeled vehicle I have three types of engine drive. So, actually I have a combination of three into three nine combinations and a there may be several live class which have the same base class parent. So, you will actually have lot more than nine explosive kind of combine multiply get the evenly explosive kind of combinations at the live level. So, this is quite in terms of modelling, but when you actually have to manage the code, and remember these whole hierarchies and like the logic on top this, deciding at every stage as to what is inherited. And you know what using you should use and what you should override this becomes not a very good aid in the design rather it becomes a hindrance.

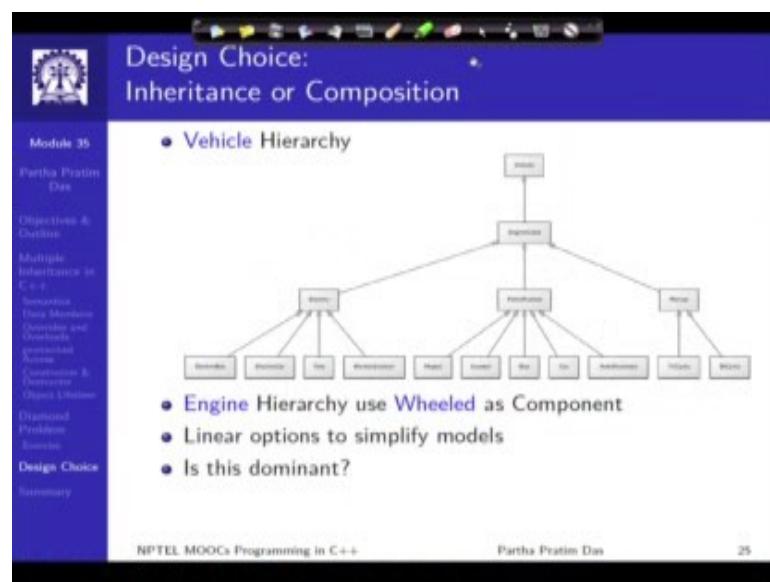
(Refer slide Time: 15:48)



So, what is a general prescription is, if you have multiple possible hierarchies in a domain then you should choose one which is the dominant one, that is one way of looking at even that may help you get rid of multiple inheritance in many places is as the vehicle hierarchy. So, I am just looking at the hierarchy of the wheeled vehicles. So, I am not looking at the engine types. So, what I will do is whenever I make a vehicle object of any derive types.

So, these are different wheeled vehicles four, three, two and then you have all the base class, for that I will have in this object I will have an engine star say pointed to engine and this will basically point to the engine class having all the different specialization. So, whether I have manual, whether I have electric, whether I have petrol fluid and so on, instead of deriving this object jointly from two hierarchies I will derive it on one here on this particularly wheel side and then refer to the engine hierarchy as a member, as a composition. So, this pointer then will lead to the particular type of engine as I want.

(Refer slide Time: 17:13)



Of course, if I am doing this, then of course I can do the other way round, I can do a vehicle hierarchy again, which is basically based on the engine class. So, I have the electric engine, petrol fuel engine, manual engine, I have my leaf label classes on that, but now in a wheel I have a wheel pointer, which basically tells me the hierarchy of two

wheeler, three wheeler, four wheeler and so on. And I primarily work on the engine type as a dominant hierarchy and use the other one HASA or component of this vehicle type.

So, either of that can be done, if there are multiple notations in the in the multiple inheritance structure that exist then you will have several such options. And the most prescribed and more commonly used style is you identify one of them, one of the different options of inheritance is a hierarchy that you are getting, take identify one of them which is dominant and put a single multi-level and you know kind of a hierarchy on that. And all the other hierarchies that you are getting all the other is a relationship like the engine one we saw here make them into their components. And then you travels that components and going to the particular hierarchy like this. And do the reasoning about the engine type on this hierarchy if you want say you want to have another hierarchy in terms of the carriage type as to whether it is for passengers, whether it is for a goods, whether it is it is for custody and so on.

So, accordingly you have to it; now this is a design choice to what is a dominant one which must be on a hierarchy. So, that you can actually write polymorphic code on that and then refer to the different components in the alternate hierarchies. And there are several design techniques by which you can make what is known as a double dispatch like jointly you can a decide on two independent hierarchies and actually dispatch a polymorphic function that will be beyond the scope of this course as you become more and more expert, you will able to learnt and see all that.

(Refer slide Time: 19:37)

The slide is titled "Module Summary" and is part of "Module 35" by Partha Pratim Das. The slide content includes a list of three bullet points summarizing the module's topics:

- Introduced the Semantics of Multiple Inheritance in C++
- Discussed the Diamond Problem and solution approaches
- Illustrated the design choice between inheritance and composition

The sidebar on the left lists various topics covered in the module, such as Objectives & Outcome, Multiple Inheritance in C++, Specialization, Data Members, Overriding and Overloading, Generalization, Association, Composition & Derivation, Object Lifetime, Diamond Problem, Review, Design Choices, and Summary.

At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" with slide number 26.

So, to summarize in two lectures in this module, we have introduced the concepts of multiple inheritance, we have explained the basic semantics. And I have tried to just give you glimpse of what are the different pitfalls that may happen, when you use multiple inheritance. And at the end up, try to give you some idea about a design choice between using inheritance as only mechanism of object structuring vis-a-vis making a mix of inheritance and composition and deciding on a dominant hierarchy of object as your major polymorphic representation of the objects. And using the other ones as composition in terms of reference and then making your polymorphic dispatch according to that.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 52
Exceptions (Error Handling in C): Part – I

Welcome to module 36 of programming in C++. In this module and the next, we will try to take a look into exceptions. what is called exceptions in C++? It is basically handling errors, errors and extreme exception situations of a system, in general. And there are varieties of options that exist for it. Actual C++ does provide a whole lot of features, a very powerful exception handling feature. Before getting into that which we do in the next module.

(Refer Slide Time: 01:04)

The screenshot shows a presentation slide with a blue header bar containing the text "Module Objectives". Below the header, on the left, is a sidebar with a logo and navigation links for "Module 36" and "Objective & Outline". The main content area contains a single bullet point: "Understand the Error handling in C". At the bottom of the slide, there is footer text: "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and a small number "2".

We would first take a quick look into the error handling options that exist in C. I am sure you are aware of these already, but we just take a look, because they remain valid in C++ as well, and but they have certain short comings which the C++ exceptions would try to wave off.

(Refer Slide Time: 01:26)

The slide title is "Module Outline". On the left, there is a vertical navigation menu for "Module 36" with sections like "Objective & Outline", "Exception Fundamentals", "Exceptions in C", "Exceptions in C++", and "Summary". The main content area lists topics under "Exception Fundamentals": "Types of Exceptions" and "Exception Stages". It also lists topics under "Exceptions in C": "C Language Features", "Standard Library Support", and "Shortcomings". Under "Exceptions in C++", it lists "Exception Scope (try)", "Exception Arguments (catch)", "Exception Matching", "Exception Raise (throw)", and "Advantages". At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

So, this is the module outline for both the module 36 as well as module 37, certainly the blue part is what we do in this current module and that will be available on the left of your screen also the slides.

(Refer Slide Time: 01:44)

The slide title is "What are Exceptions?". On the left, there is a vertical navigation menu for "Module 36" with sections like "Objective & Outline", "Exception Fundamentals", "Exceptions in C", "Exceptions in C++", and "Summary". The main content area lists two bullet points: "Conditions that arise" (with sub-points about infrequent errors, program termination, and run-time anomalies) and "Leading to" (with sub-points about program crippling, system failure, defensive techniques, and exception handling design). At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

So, the question is, what are exceptions? The exceptions are conditions that arise usually infrequently and certainly unexpectedly in many places. It generally betrays a program error and requires a certain amount of programmatic response. Usually, exceptions these conditions arise due to run time anomalies, but it may not necessarily be just run time anomalies they could occur even otherwise. And when these errors happen then that lead to crippling of the program. So, the program crashes or gets into some indeterminate state and things like that. And if it is really bad then it could not only impact the current program, but it could pull the system down all together.

So, certainly in C, what we have been doing is we have been following several defensive techniques that try to conceive all different possible negative parts, all possible situations where things might go wrong and then try to take care of them. So that what happens is, when we try to take care of the exceptions in this way then we get tangled into a whole lot of error code rather the code that needs to take care of these error situations, you get tangled into a whole lot of error paths, which clutter the basic design paths in the program. So that is a basic issue of exceptions situation.

(Refer Slide Time: 03:18)

The slide has a blue header with the title 'Exception Causes'. On the left, there is a sidebar with navigation links: 'Module 36', 'Partha Pratim Das', 'Objective & Outline', 'Exception Fundamentals', 'Types of Exceptions', 'Exception States', 'Exceptions in C', 'Language Features', 'File I/O Examples', 'Local vars', 'C Standard Library', 'Memory Leaks', 'Conclusion', and 'Q & A'. Below the sidebar is a circular profile picture of a man. The main content area contains a bulleted list of causes:

- Unexpected Systems State
 - Exhaustion of Resources
 - Low Free Store Memory
 - Low Disk Space
 - Pushing to a Full Stack
- External Events
 - C
 - Socket Event
- Logical Errors
 - Pop from an Empty Stack
 - Resource Errors – like Memory Read/Write
- Run time Errors
 - Arithmetic Overflow / Underflow
 - Out of Range
- Undefined Operation
 - Division by Zero

And there are several causes I mean this is an indicative lecture to which covers most kinds of exceptions, but there could others. For example, primary being the unexpected

systems state, for example, exhausting resources like memory, disk space and so on. Like trying to push into a stack whose internal store is already become full. It could be due to external events like program termination given by the user control C or something like that. It could be due to a socket event, or it could also be due to different logical errors, for example, logical error is one like you are trying to pop from an empty stack.

Now, this is not related to system resources, but logically this is an error. There could be resource errors like memory read, write errors and so on. They could be other run time errors like arithmetic overflow underflow prior to add two numbers and that overflows the number it could be out of range for this. And it could also be for undefined operations like division by 0 is something which will lead to exceptions. So, there are these are some of the typical reasons, but there could be more of those.

(Refer Slide Time: 04:35)

The slide has a blue header with the title 'Exception Handling?'. On the left, there's a sidebar with a logo, the text 'Module 36', 'Partha Pratim Das', 'Objective & Outline', 'Exception Fundamentals', 'Types of Exceptions', 'Exception Stack', 'Exceptions in C', and a small image of a person. The main content area contains a bulleted list and a quote:

- Exception Handling is a mechanism that separates the detection and handling of circumstantial Exceptional Flow from Normal Flow
- Current state saved in a pre-defined location
- Execution switched to a pre-defined handler

Exceptions are C++'s means of separating error reporting from error handling
– Bjarne Stroustrup

So, exception handling is what is very important in terms of C++ as Bjarne Stroustrup's comment shows, it is a mechanism that separates, this is the key idea that it tries to separate the circumstantial exceptional flow that is flow that arise due to these exceptional situation happening from the normal or the happy flow that we have actually designed for. So, that is basic thing that we want to do understand here. So, if the current state in this case should be saved in a pre defined location and the execution of the

program should be handed over to a predefined exception handler or a function or a piece of code which is supposed to take care of the error situation.

(Refer Slide Time: 05:28)

The screenshot shows a presentation slide with a blue header containing the title 'Types of Exceptions'. The slide content is organized into two main sections: 'Asynchronous Exceptions' and 'Synchronous Exceptions'. Under 'Asynchronous Exceptions', there are three bullet points: 'Exceptions that come Unexpectedly', 'Example - an Interrupt in a Program', and 'Takes control away from the Executing Thread context to a context that is different from that which caused the exception'. Under 'Synchronous Exceptions', there are three bullet points: 'Planned Exceptions', 'Handled in an organized manner', and 'The most common type of Synchronous Exception is implemented as a `throw`'. The slide footer includes the text 'NPTEL MOOCs Programming in C++', 'Partha Pratim Das', and a page number '7'.

So, if we also look at what are the types of exceptions? We can see that of the different causes that we have just listed. They can be broadly classified into two groups the asynchronous and the synchronous. Asynchronous is which is happening from a different thread usually. So, it is not in concurrence with the normal flow of the program that comes unexpectedly like the interrupt in a program and so on. Or it could be a planned exception planned in the sense; of course you do not plan for errors, but planned in the sense that it is something that happens along the flow of the program.

For example, you wanted to allocate memory and the resource was low, so an exception happens, you wanted to divide by a number and that number turns out to be zero. So there will be little bit of different style to handle asynchronous exception, but for majority of the situations which are synchronous in nature. We would be like to that is the common situation. We will be implementing them in terms of certain feature in C++ called a throw feature.

(Refer Slide Time: 06:36)

The slide is titled "Exception Stages" and is part of Module 36. It features a sidebar with navigation links for various topics like Objective & Outline, Exception Fundamentals, Types of Exceptions, and Exception Stages. The main content area is titled "Exception Stages" and lists five stages:

- ① Error Incidence**
 - Synchronous (S/W) Logical Error
 - Asynchronous (H/W) Interrupt (S/W Interrupt)
- ② Create Object & Raise Exception**
 - An Exception Object can be of any Complete Type
 - An int to a full blown C++ class object
- ③ Detect Exception**
 - Polling Software Tests
 - Notification Control (Stack) Adjustments
- ④ Handle Exception**
 - Ignore: hope someone else handles it, that is, Do Not Catch
 - Act: but allow others to handle it afterwards, that is, Catch, Handle and Re-Throw
 - Own: take complete ownership, that is, Catch and Handle
- ⑤ Recover from Exception**
 - Continue Execution: If handled inside the program
 - Abort Execution: If handled outside the program

At the bottom, there is a footer with "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

We will look into those. Now, before we get into the analysis, let us understand that exception actually happens in five different stages let me just first show you an example then we can come back here.

(Refer Slide Time: 06:48)

The slide is titled "Exception Stages" and is part of Module 36. It features a sidebar with navigation links for various topics like Objective & Outline, Exception Fundamentals, Types of Exceptions, and Exception Stages. The main content area is titled "Exception Stages" and shows the following C++ code illustrating the five stages:

```
int f() {
    int error;
    /* ... */
    if (error) /* Stage 1: error occurred */
        return -1; /* Stage 2: generate exception object */
    /* ... */
}

int main(void) {
    if (f() != 0) /* Stage 3: detect exception */
    {
        /* Stage 4: handle exception */
    }
    /* Stage 5: recover */
}
```

At the bottom, there is a footer with "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

This is a very typical situation, this a function main which is calling a function f and in this case, it is assumed that as if, if error, function f works correctly then it will returns a 0; otherwise, it returns a non-zero value. So, you check if f has returned 0. And if it is not returned 0, then you know that something has gone wrong and you try taking care of that. So, if you look into what can happen is f is executing has been called is executing, let say is executing, and at some point it gets into a error situation this is I have just shown it as a as a kind of bool here, but this could arise due to many situations like low memory and so on. And this is called the stage one of an exception where you identify that an error situation has happened.

And then you report this error because, for example, if it is low memory f does not know what to do in case of low memory. So, here f will have to report that which we say you generate an exception object, you will understand what does that mean in the context of C++ more, but basic thing is you want to report the error to the caller. So, when that happens the control comes back here, and control comes back with minus one which certainly fails this certainly satisfies this test. So, you know that an exception has happened that an error situation has happened.

So, the third stage is detect that the caller has been able to detect that an error has happened. And then this is what we say is a handler is a piece of code which should be executed only when this error situation as happened. For example, if this is due to low memory, then you may want to decide what you want to do here. For example, you could realize some of the memory that you have already allocated and you are not using at present you could release them so that more memory becomes available and you can go back and call f again. Whatever you decide to do is the basic handling of the exception. And once that has been handled then you have back to the normal flow which you say is a recover phase that is you are recovered from that error.

So, create, report, detect, handle, and recover are the basic five stages through any exception can be handled. So, if we just go back to the; I am sorry, want to go back to the previous slide. So, here these are five stages error incidence, which is create, this is create object, and rise which is report, this is the detect, the handle and the recover which other different things that you can do in case of an error situation. And how you do that

depends on the mechanism that you are that is available to you mechanism that you are using.

(Refer Slide Time: 09:46)

The slide has a blue header with the title 'Support for Exceptions in C*' and the NPTEL logo. The main content area is titled 'Module 36' and 'Partha Pratim Das'. It includes a sidebar with 'Objective & Outline' and a list of topics. The main content lists 'Language Features' and 'Standard Library Support' with their respective sub-points.

Module 36	Partha Pratim Das
Objective & Outline	
Exception Fundamentals	
• Basic Concepts	
• Exception States	
Exceptions in C	
• Language Features	
• SV & Progress	
• Standard Library Support	
• Standard Library Functions	

- Language Features
 - Return Value & Parameters
 - Local goto
- Standard Library Support
 - Global Variables
 - Abnormal Termination
 - Conditional Termination
 - Non-Local goto
 - Signals

NPTEL MOOCs Programming in C++ Partha Pratim Das 10

So, in C, you have a variety of mechanisms and only very little is supported by the language actually the language does not support anything the C language does not support anything keeping error handling in mind. But you can use some of the language features to handle errors in a more structured way and instead, there are several libraries which is the part of the standard library which give you additional feature to be able to handle those errors. So, the first that you can do we will just take a quick look.

(Refer Slide Time: 10:18)

The screenshot shows a presentation slide with a blue header bar containing the title 'Return Value & Parameters'. Below the header is a sidebar with a list of topics: Module 36, Partha Pratim Das, Objective & Outline, Exception Fundamentals, Types of Exceptions, Execution Stages, Exceptions in C, C Language Features, RV & Params, Local vars, C Standard Library Functions, Under Construction, and Help. The main content area contains two bullet-pointed sections: 'Function Return Value Mechanism' and 'Function (output) Parameter Mechanism'. The 'Function Return Value Mechanism' section includes points about creation by the callee as temporary objects, passing onto the caller, caller checks for error conditions, return values being ignored or lost, and return values being temporary. The 'Function (output) Parameter Mechanism' section includes a point about outbound parameters bound to arguments offering multiple logical return values. At the bottom of the slide, there is a footer with the text 'NPTEL MOOCs Programming in C++', 'Partha Pratim Das', and the number '13'.

This is something, which we are all familiar with that I call a function and I expect that function to return a value and that value will denote whether the function has successfully completed or there is an error, so this is by return value mechanism. Now if your function is actually suppose to return a value based on computation then you may want to return this error condition in terms of an additional parameter, possibly we will have some kind of a call by address parameter where you put the error and put it back. Now, the basic problem of these kind of error handling is, after the error has been reported it needs to be checked by the caller because if the error is a reported, but then the caller does not detect it then it carries on in some indeterminate stage. So, a return values could be lost could be ignored and so that this is not a very effective mechanism for doing this.

(Refer Slide Time: 11:26)

```
int Push(int i) {
    if (top_ == size-1) // Incidence
        return 0; // Raise
    else
        stack_[++top_] = i;
    return 1;
}

int main() {
    int x;
    // ...
    if (!Push(x)) // Detect {
        // Handling
    }
    // Recovery
}
```

But several C programs use these mechanisms, for example, here I am looking at a push which returns an int. You will recall we have looked at push method for stack very often, and we typically have a void in the return. Because we do not really expect anything back from the push, but just to take care of the error you could have that it returns an int and so if it is goto, say the store has become full it returns 0 to designate that the push has failed. So, the caller will have to again detect that and handle that further. So, these are very simple mechanism very widely used a lot of C library is use this, but quite inefficient in that matter.

(Refer Slide Time: 12:10)

The slide has a blue header bar with the title 'Local goto'. On the left, there is a sidebar with a navigation menu for 'Module 36' containing items like 'Partha Pratim Das', 'Objective & Outline', 'Exception Fundamentals', 'Types of Exceptions', 'Inception Stages', 'Exceptions in C', 'C Language Features', 'BY & Partha', 'local goto', 'C Standard Library Functions', 'Global Variables', 'About Author', and 'Feedback'. Below the sidebar is a video player showing a man in a white shirt. The main content area contains two bullet points:

- Local goto Mechanism
 - (At Source) Escapes: Gets Control out of a Deep Nested Loop
 - (At Destination) Refactors: Actions from Multiple Points of Error Inception
- A group of C Features
 - goto Label;
 - break; & continue;
 - default switch case

At the bottom of the slide, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

Another that we that is very frequently uses a local goto. The example will I mean there are there are several ways you can you can do local goto, in terms of actual goto, break continue, default switch case, these are all cases. For example, in switch we have a default. So, these are all cases where we are trying to take care of situations we have not been able to take care in the happy path. So, local goto are a very typical way to handle.

(Refer Slide Time: 12:39)

The slide has a blue header bar with the title 'Example: Local goto'. On the left, there is a sidebar with a navigation menu for 'Module 36' containing items like 'Partha Pratim Das', 'Objective & Outline', 'Exception Fundamentals', 'Types of Exceptions', 'Inception Stages', 'Exceptions in C', 'C Language Features', 'BY & Partha', 'local goto', 'C Standard Library Functions', 'Global Variables', 'About Author', and 'Feedback'. Below the sidebar is a video player showing a man in a white shirt. The main content area contains a block of C code:

```
_PHNDLR _cdecl signal(int signum, _PHNDLR sigact)
{ // Lifted from VC98\CRT\SRC\WINSIG.C
... /* Check for sigact support */
    if ( (sigact == ...) ) goto sigreterror;

    /* Not exceptions in the host OS. */
    if ( (signum == ...) ) { ... goto sigreterror; }
    else { ... goto sigretok; }

    /* Exceptions in the host OS. */
    if ( (signum ...) ) goto sigreterror;
...
sigretok:
    return(oldsigact);

sigrerror:
    errno = EINVAL;
    return(SIG_ERR);
}
```

At the bottom of the slide, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

For example, I have this code which, please do not bother about what the code does, but just look at fact that there are two labels used here. So, these two labels are designative of the successful functioning of the code and error functioning of the code. So, in different parts of the code, wherever you have an error situation happening say like this, you basically jump to this label; whereas, if you have a successful completion then you jump to this. What helps you there is in case of error possibly you will have lot of code to take care of lot of things to maybe certain objects which need to be deleted which need to be destructed and so on, all these you can take care of at one place you do not need to copy that code it multiple places.

(Refer Slide Time: 13:30)

The screenshot shows a presentation slide with the title "Example: Local goto". The slide content is a block of C code:PHNDLR _cdecl signal(int signum, _PHNDLR sigact)
{ // Lifted from VC98\CRT\SRC\WINSIG.C
... /* Check for sigact support */
if ((sigact == ...)) goto sigreterror;

/* Not exceptions in the host OS. */
if ((signum == ...)) ... goto sigreterror;
else { ... goto sigretok; }

/* Exceptions in the host OS. */
if ((signum ...)) goto sigreterror;
...
sigretok:
 return(oldsigact);

sigreterror:
 errno = EINVAL;
 return(SIG_ERR);
}

The slide also includes a sidebar with course navigation links and a small video thumbnail of the speaker.

So, this is how it would look like that you have this code and in error situations in all these it comes to this level; whereas if you have successful completion it comes to the sigretok label and it returns from there. So, this is just one convenient way to take care of errors.

(Refer Slide Time: 13:48)

The screenshot shows a presentation slide titled "Global Variables". The slide content is as follows:

- GV Mechanism
 - Use a designated Global Error Variable
 - Set it on Error
 - Poll / Check it for Detection
- Standard Library GV Mechanism
 - <errno.h> / <cerrno>

The slide has a blue header bar with the title "Global Variables". On the left, there is a sidebar with a navigation menu and a small portrait of a man. The main content area contains the bulleted list. At the bottom, there is footer text: "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" along with a page number "18".

Another which is known as global variables. So, in many cases since a c has the functions as the global and there is no other scoping at that way. So, you may unrelated functions particularly the library functions may not have way to designate how the error will happen. So, what they do is they basically set a global variable if there is an error and this is given in the standard library errno.h which in c it will become cerrno.h as you know.

(Refer Slide Time: 14:25)

Module 36
Partha Pratim Das

Objective & Outcome
Execution Fundamentals
Types of Exceptions
Exception Handling
Exceptions in C
C Language Features
UV & Macros
Local goto
C Standard Library Functions
Global Variables

NPTEL MOOCs Programming in C++
Partha Pratim Das 19

```
#include <errno.h>
#include <math.h>
#include <stdio.h>

int main() {
    double x, y, result;
    /*... somehow set 'x' and 'y' ...*/
    errno = 0;

    result = pow (x, y);

    if (errno == EDOM)
        printf("Domain error on x/y pair \n");
    else if (errno == ERANGE)
        printf("range error in result \n");
    else
        printf("x to the y = %d \n", (int) result);

    return 0;
}
```

So, in this for example, here we are trying to do a power, this is a pow is a library function which rises x to the power y and it might become just too large. So, if it does become too large, then it sets a variable error no. This error no you can see that it is declare here because this is declared as a global. So, you start by setting it to 0, which is basically clearing it out saying that there is no error. And then if this function gets into some range error x to the power y may be just too large to represent then error no is set to erange which is a manifest constant defined again in the library which tells you that this condition as happened.

If it is a domain error that is if is something on which you cannot rise the power x to the power y then it errno will become edom. So, again you are responsibility, the basic create and report is within this library function pow, and as an application programmer, it is your responsibility to detect that you can see that ah code which otherwise would have been just this much has become so much has become, so that you can actually take care of the error situation. But using global variables like this, this header is a very common way to handle errors in C.

(Refer Slide Time: 15:43)

The slide is titled "Abnormal Termination". On the left, there is a sidebar with the title "Module 36" and a list of topics: Partha Pratim Das, Objective & Outcome, Execution Fundamentals, Types of Exceptions, Exception Stage, Exceptions in C, C Language Features, I/O & Streams, Local scope, C Standard Library, Standard Global Headers, and NTPTEL. Below the sidebar is a video player showing a man speaking. The main content area contains a bulleted list of functions for abnormal termination:

- Program Halting Functions provided by
 - <stdlib.h> / <cstdlib>
- **abort()**
 - Catastrophic Program Failure
- **exit()**
 - Code Clean up via atexit() Registrations
- **atexit()**
 - Handlers called in reverse order of their Registrations

At the bottom of the slide, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" with a page number "20".

Several errors particularly mean that we cannot proceed further than that. So, if that error has happened then we need to actually terminate the program. Now, there are two kinds of termination that typically C supports; the major part is abnormal termination, which is for this several functions have provided in the C standard library. One is abort where which you can call at any point in a C program and hence in a C++ program. So, this is known as a catastrophic program failure. So, what abort does is that from that point exactly from that point where abort is called, it simply makes and exit from the function not only that function from the whole program all together. However, deeply it may be nested and just returns it to the environment.

In contrast, you could also use exit function, the abort and exit in terms of termination are both abnormal terminations, they have similar effects in that way. But there is a significant difference in terms of the fact that if there are say global objects which have been created initialized, there are local objects which have been initialized, and you are at a point where you want to basically terminate an exit. If you abort then it is catastrophic, so you do not call the destructors of these objects.

So, these objects basically exit without destructing, these objects which could otherwise be very dangerous. For example, a one of the global objects are created could be a single

ton which is holding a say a lock to a global database. And if you do not call the destructor which is suppose to release that lock it means that the when your program terminates the lock is still held by your program, and it is very difficult to unlock that in future.

In contrast, exit also has a similar behavior, but it does a code clean up, it basically calls all the destructors which have been registered so far in the reverse order of their construction and calls them one after the other, so it does clean up. And for doing this clean up it uses another library function called atexit; atexit is basically a handler common handler for all these errors. So, what atexit does it, it basically registers different destructors so as function pointers and when you exit from the program it will call them in the reverse order of their actual creation.

(Refer Slide Time: 18:18)

The screenshot shows a presentation slide with a blue header bar containing the title 'Example: Abnormal Termination'. Below the header is a sidebar with a list of module topics: Module 36, Partha Pratim Das, Objective & Outline, Exception Fundamentals, Loss of Control, Exception Signals, Exceptions in C, C Language Features, Partha Pratim Das, Standard Library Support, and Abnormal Termination. The main content area displays the following C code:

```
#include<stdio.h>
#include<stdlib.h>
static void atexit_handler_1(void) {
    printf("within 'atexit_handler_1'\n");
}

static void atexit_handler_2(void) {
    printf("within 'atexit_handler_2'\n");
}

int main(){
    atexit(atexit_handler_1);
    atexit(atexit_handler_2);
    exit(EXIT_SUCCESS);

    printf("This line should never appear\n");

    return 0;
}
/* On Execution Output: within 'atexit_handler_2'
   within 'atexit_handler_1'
   and returns a success code to calling environment */
```

At the bottom of the slide, there is a footer with the text 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

So, you could try out this code, which show you that you have two handlers defined here. Do not worry about what this handlers do, they practically do nothing and in the main you could register them by with the atexit, so that if you exit from the program here you are doing you are basically doing successful exit. If you do any exit from the program then it does not immediately exit, it will first call the last register function, which is atexit handler 2 then it will call the earlier register function atexit handler 1 and only

after that it will exit. So, this mechanism can be used not only to clean up the just a clean up the objects by calling their destructor, but it can also be used to perform any clean up task that you may want and you can register appropriate handler for that with atexit and then exit with success or with failure whatever values you want. So, abnormal termination is a major feature which at least allows you to stop a program crash of course, if you are not using abort, abort is basically equivalent to a crash in the program.

(Refer Slide Time: 19:33)

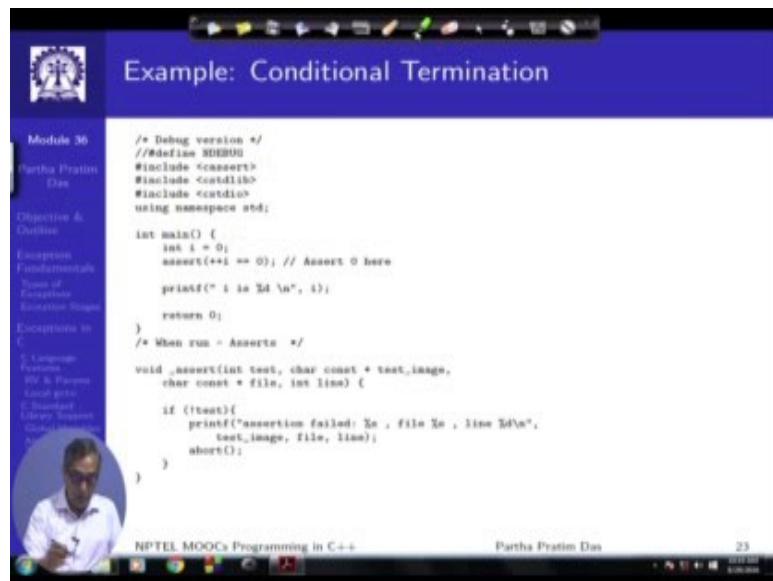
The screenshot shows a presentation slide with a blue header bar containing the title 'Conditional Termination'. On the left side, there is a vertical sidebar with the following navigation links: Module 36, Partha Pratim Das, Objective & Outcome, Exception Fundamentals, Types of Exceptions, Exception States, and Exceptions in C++. Below these links is a circular profile picture of a man. The main content area contains a bulleted list of points about the assert macro:

- Diagnostic ASSERT macro defined in
 - <assert.h> / <cassert>
- Assertions valid when NDEBUG macro is not defined (debug build is done)
- Assert calls internal function, reports the source file details and then Terminates

At the bottom of the slide, there is footer text: NPTEL MOOCs Programming in C++, Partha Pratim Das, and the number 22.

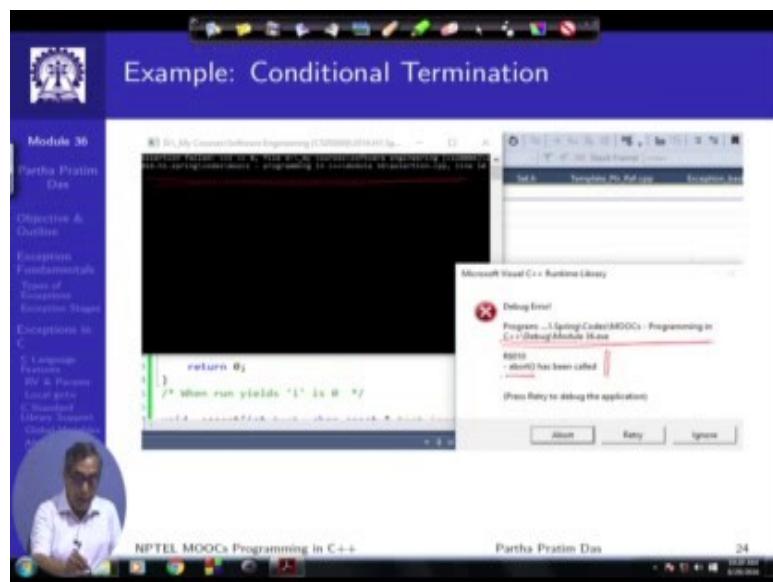
So, another that you can use which is called a conditional termination, this is basically a debug time feature, there is a library called assert which has a assert macro, which you can provide to assert a certain condition in terms of the program. So, I will just show you an example.

(Refer Slide Time: 19:51)



So, what you are doing here is, int i is 0; and then you are asserting that ++ i is equal to 0. We are saying that i must be asserting this means this condition must be true. So, if this condition is true then it does not do anything; if this condition is false then it raises an exception then it will throw and a certain kind of window out and tell you that this has happened. So, if you actually run this program then I will show what happens.

(Refer Slide Time: 20:24)



If we run this program it will show you something like this that it says that at this while running this program here, it is not very visible, but you can actually see what is the program source and in which line it has happened. It says that your assert has happened and result of assert is actually calling abort, but this helps you to check different conditions at run time in a program, and check out, let me just go back to this. So, it helps you to check out if this condition where satisfied, here obviously this is a stupid code. So, this condition is not supposed to be satisfied.

Now, the advantage of assert is certainly if you put too many of them in the program then at the run time there will be lot of places where your checks will keep on happening and that will become a detrimental to the performance of the program. So, what assert gives you is, there is a manifest constant and debug. In some compiler, it is called a in debug; in some other compiler, it will be called something else. So, if that is this means that you are not debugging. So, if you put that on then this assert feature would not be there. So, if you build a debug version which I did, I have actually commented out this. So, I am saying that it is indeed debug version. So, assert must assert.

But if I put this on, I will just show you, if I put this on, I have now put this on I making a release version where I do not want this assert would be there which will fail and all that, but it is just that it will not assert and report me anything. Or in other words, basically this assert at the compile time will get removed in the build code the build code will actually not have that assert because this is done based on the n debug. So, this is another way that you can do a conditional termination and particularly very useful while you are debugging your program.

(Refer Slide Time: 22:28)

The screenshot shows a presentation slide with a blue header bar containing the title 'Non-Local goto'. Below the header is a sidebar with a logo and a list of navigation links. The main content area contains a bulleted list of functions related to setjmp and longjmp.

Module 36
Partha Pratim Das
Objective & Outline
Exception Fundamentals
Types of Exceptions
Exception Scope
Exceptions in C
C Language Features
HW & Programs
Local goto
C Standard Library Functions
Global Variables

Non-Local goto

- `setjmp()` and `longjmp()` functions provided in `<setjmp.h>` Header along with collateral type `jmp_buf`
- `setjmp(jmp_buf)`
 - Sets the Jump point filling up the `jmp_buf` object with the current program context
- `longjmp(jmp_buf, int)`
 - Effects a Jump to the context of the `jmp_buf` object
 - Control return to `setjmp` call last called on `jmp_buf`

NPTEL MOOCs Programming in C++ Partha Pratim Das 26

Last feature which many of you may not be very familiar with this is called a non-local goto. For which you have two functions provided in a standard library header called `setjmp`, `setjmp.h`, which has two functions set jump and long jump. And it takes a jump buffer as a parameter. So, what you basically do is ah certainly the local goto's are are a good mechanism, but certainly you cannot have vocal goto's across functions because as you know that goto's are always limited to one function scope. So, non-local goto's are set jump, long jump give you a mechanism by which you could have jumps across functions.

(Refer Slide Time: 23:16)

The slide title is "Example: Non-Local goto: The Dynamics". The left sidebar contains course navigation links: Module 36, Partha Pratim Das, Objective & Outcome, Execution Fundamentals, Trace of Exceptions, Execution Steps, Exceptions in C, C Language Features, HW & HWs, Local point, Standard Library Functions, Under Construction, and Help. A video thumbnail of the speaker is also present.

Caller	Callee
<pre>void f() { A a; if (setjmp(jbuf) == 0) { B b; g(); h(); } else { cout << ex.what(); } return ; }</pre>	<pre>jmp_buf jbuf; void g(){ A a; UserExcpt ex ("From g()"); longjmp(jbuf, 1); return ; }</pre>

At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" with slide number 27.

So, this is how you write it suppose f is a function, and g is another function, and f basically calls g. So, what you do is you define a buffer jump buffer say j buff, and in the program in the function being called you could give a long jump with the buffer with certain integer code. So, what will happen is in the caller, if you have called set jump then when the function control returns, it will return either with this value if you have executed this long jump which is an error condition possibly or it will return with 0. So, if returns with a 0, you know that this is a gone in without an error; if it returns with a 1, you know that at this point you had the error.

Now the significance of this number 1 is certainly you could have multiple of them in your called function. I can have a j buff at a different point with 2. So, if I fail from that point then when I come back to the caller with this set jump check, I will actually find this value 2. So, this is this some point is called 1, some point is being called 2, some point may be called 3 and default is 0. So, if I have a normal termination, set jump will come back with 0. So, I will be able to just continue with normal execution, but if I come out with any of these error points then set jump will have an appropriate value of whatever you have long jumped with, and based on that you could write the else conditions. For example, here I just show with 1, so I have a 1, so if set jump comes

here, then the normal condition it will have a 0; if it is executed long jump, it will come with 1, so the else will get satisfied where you can find out what situation is.

(Refer Slide Time: 25:19)

```
void f() {
    A a;
    if (setjmp(jbuf) == 0)
    {
        B b;
        g();
        h();
    }
    else {
        cout << ex.what();
    }
    return;
}

jmp_buf jbuf;

void g()
{
    A a;
    UserExcp ex("From g()");
    longjmp(jbuf, 1);
    return;
}
```

- g() called

So, if you just look it in terms of dynamics then this is what the g is getting called.

(Refer Slide Time: 25:25)

```
void f() {
    A a;
    if (setjmp(jbuf) == 0)
    {
        B b;
        g();
        h();
    }
    else {
        cout << ex.what();
    }
    return;
}

jmp_buf jbuf;

void g()
{
    A a;
    UserExcp ex("From g()");
    longjmp(jbuf, 1);
    return;
}
```

- g() successfully returns

So, in normal situation g successfully completed, this did not get executed this was not executed. So, you return you come back here, and continue you come back to h because that is a statement immediately following g.

(Refer Slide Time: 25:41)

```

Module 30
Partha Pratim Das
Objective & Outline
Exception Fundamentals
    - Exception Handling
    - Exception Signals
Exceptions in C
    - Language Features
    - PV & Threads
    - Local goto
    - C Standard Library Functions
    - Global Functions
    - Assertions

```

```

void f() {
    A a;
    if (setjmp(jbuf) == 0)
    {
        B b;
        g();
        h();
    }
    else {
        cout << ex.what();
    }
    return;
}

jmp_buf jbuf;

void g()
{
    A a;
    UserException("From g()");
    longjmp(jbuf, 1);
}

```

- g() called and longjmp() executed
- setjmp() takes to exception part

But instead if you called g, and you come across this that is this under some condition that this error situation has happened in g, you do a long jump 1, then also you come back. As soon as you do long jump, you actually control actually comes back it does not wait to go up to the return, because this is some kind of error that you have come across. It does come back, but it does not come back to h because you have not completed g. It comes back to check what the set jump value is set jump value is 1 here. So, this condition has failed, so it comes back to else.

Here, you could have if set jump j buf equal to equal to 1, else something. So, it depending on that you can have a switch in the calling function f to decide which particular point g has failed from. So, this is the less known mechanism and we will see that of course, this is not very clean mechanism this is we will need remember how many points of function can come back from and obviously, everything will have to be pre decided pre designed and so on.

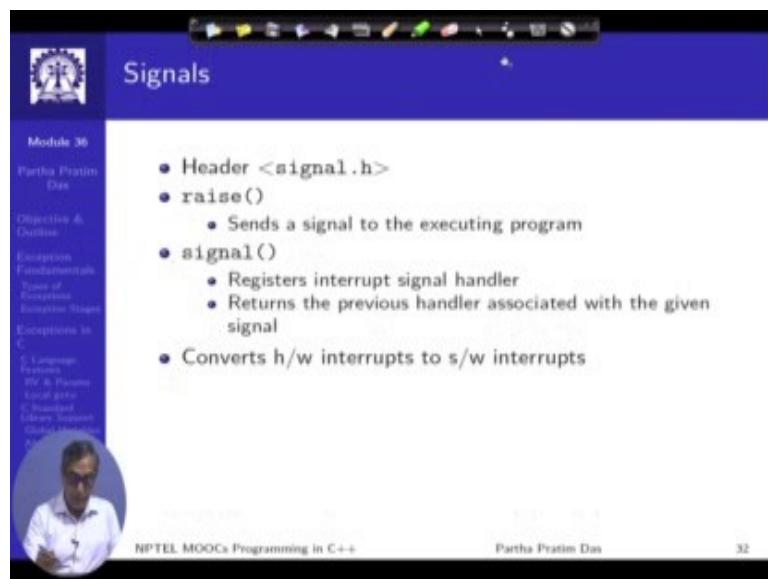
(Refer Slide Time: 26:43)

The screenshot shows a video player interface with a blue header bar. The title 'Example: Non-Local goto' is displayed in white text. Below the title, there is a navigation bar with icons for back, forward, search, and other controls. The main content area contains a C code snippet. On the left side of the slide, there is a sidebar with a circular profile picture of a man, the text 'Module 36', and a list of course topics. At the bottom of the slide, there is footer text: 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

```
#include<setjmp.h>
#include<stdio.h>
jmp_buf j;
void raise_exception(){
    printf("Exception raised \n");
    longjmp(j, 1); /* Jump to exception handler */
    printf("This line should never appear \n");
}
int main(){
    if (setjmp == 0) {
        printf(" 'setjmp' is initializing j \n");
        raise_exception();
        printf("This line should never appear \n");
    }
    else
        printf(" 'setjmp' was just jumped into \n");
        /* This code is the exception handler */
    return 0;
}
/* On execution : 'setjmp' is initializing j ,
exception raised and 'setjmp' was just jumped into */
```

This here I have given a code which you can just copy paste and try out this behavior of set jump long jump.

(Refer Slide Time: 26:52)



Besides that, you have signals which people who have done some programming in terms of operating system will know there is mechanism to send signal. You can send a signal

this is in signal dot h and there is a associated handler, that is your function pointer which you can put.

(Refer Slide Time: 27:09)

The screenshot shows a presentation slide titled "Example: Signals". The slide has a blue header with the title and a small logo. The main content area contains a C program. The code uses the `signal` function to attach a signal handler to the `SIGABRT` signal. The handler prints a message and then calls `abort()`. The code is as follows:

```
// Use signal to attach a signal
// handler to the abort routine
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <tchar.h>
void SignalHandler(int signal) {
    printf("Application aborting...\n");
}

int main() {
    typedef void (*SignalHandlerPointer)(int);
    SignalHandlerPointer previousHandler;
    previousHandler = signal(SIGABRT, SignalHandler);
    abort();
    return 0;
}
```

The slide also includes a sidebar with navigation links and a small video thumbnail of the speaker.

So, signal handlers are of this kind. So, you can define a handler and you can give different signals. For with different signals you can associate different handler, so which says that. If I have `SIGABRT` then this particular signal handler which is this one will get execute, will get invoked so that is a basic signaling mechanism and after that you may decide to abort. So, that is another mechanism available in C.

(Refer Slide Time: 27:40)

The screenshot shows a presentation slide with a blue header bar containing the title 'Shortcomings'. The slide content is organized into a sidebar and a main area. The sidebar on the left contains a logo, the text 'Module 36', 'Partha Pratim Das', 'Objective & Outcome', 'Exception Fundamentals', 'Types of Exceptions', 'Exception Stack', 'Exceptions in C', 'C Language Features', 'RV & GV', 'Local goto', 'C Standard Library Functions', 'Memory Leaks', and 'Bugs'. Below the sidebar is a video frame showing a man speaking. The main area lists four categories of shortcomings with bullet points:

- **Destructor-ignorant:**
 - Cannot release Local Objects i.e. Resources Leak
- **Obtrusive:**
 - Interrogating RV or GV results in Code Clutter
- **Inflexible:**
 - Spoils Normal Function Semantics
- **Non-native:**
 - Require Library Support outside Core Language

At the bottom of the slide, there is footer text: 'NPTEL MOOCs Programming in C++', 'Partha Pratim Das', and '34'.

So, we have seen that C has provided I mean because C did not take care of the errors situation the exception situation in the language designed, it come mostly as a post after thought. So, there is several different kinds of mechanism was plugged in then through the standard libraries and variety of them a mixture of that, but none of them is gives you a clean solution, and these are some of the short comings of error handling in C.

It is destructor ignorant that is most of these will not when you exist, when you come out of a function already scope based on exit on a terminate the program, the destructor of the currently residing objects in different local scopes will not get destructed, so that is a that is a major resource leak issue that we have. It is obtrusive in interrogating the return value or global results in a certainly a lot of code clutter is inflexible, because it is spoils the normal function semantics as we saw in push the normal semantics is not return anything. But just to take care of the error, we had to put something and this situation will get really, for example, how do you handle error in top, for example, you remember the top.

So, top is top is supposed to say it is an integer stack then top is suppose to return you an int. So, if this is the function signature, how would you return the error because you do not have error value. So, either use a global value which is a clutter which cannot happen

if you are in a recursive call. Otherwise, you pass in a parameter through which you will return that error value which is really inflexible, it changes the basic signature of the methods that we have. And it is non-native in the way that the language did not take cognizance of the error situations and is not a part of the core. So, these are some of the major difficulties of error handling in C. I just wanted to take you through this in spite of the fact that I am sure all of you have used some or all of these methods of error handling at different stages, but I just wanted to highlight that all of these really does not solves the core problem of having extreme situations having error situations.

(Refer Slide Time: 29:54)

Module Summary

Module 36
Partha Pratim Das
Objective & Outline
Exception Fundamentals
Types of Exceptions
Exception Stages
Exceptions in C++
C Language Features
HW & HWs
Local goto
C Standard Library Functions
Memory Management

- Introduced the concept of exceptions
- Discussed exception (error) handling in C
- Illustrated various language features and library support in C for handling errors
- Demonstrated with examples

NPTEL MOOCs Programming in C++
Partha Pratim Das
35

To summarize, we have introduced the concept of basic concept of exceptions, the types and stages and discuss the error handling in C. We have illustrated various language features and not really many of them, and the library support that exist in C for this, and we demonstrate it with examples. In the next module, we will get into the core of exceptions in C++.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 53
Exceptions (Error Handling in C): Part – II

Welcome to Module 37 of Programming in C++. We have been discussing about Error Handling in C - C++ in the earlier module. We have taken a look into what are the different exception causes are, variety of them and their types asynchronous and synchronous.

We then took a detailed view in terms of the different mechanism that is available in C for handling errors, handling exception. And we saw that there is hardly any mechanism available, actually no mechanism was provided in C with the thought of handling errors, handling exception situations, but the return value and non local and local goto has been used for exception handling. But there are several standard library features that what added through multiple standard libraries to provide support for handling errors in C and that still leads to a lot of a short comings lot of difficulties and in view of this here we are interested to study about error handling in C++.

(Refer Slide Time: 01:20)

The slide is titled "Module Outline". It features a sidebar with the IIT Kharagpur logo and navigation links for "Module 37", "Partha Pratim Das", "Objective & Outline", "Exceptions in C++", "Exception Scope (try)", "Exception Arguments (catch)", "Exception Matching", "Exception Raise (throw)", and "Advantages". A small portrait of Prof. Partha Pratim Das is also present in the sidebar.

The main content area is titled "Module Outline" and contains the following bullet points:

- Exception Fundamentals
 - Types of Exceptions
 - Exception Stages
- Exceptions in C
 - C Language Features
 - Return value & parameters
 - Local goto
 - C Standard Library Support
 - Global variables
 - Abnormal termination
 - Conditional termination
 - Non-local goto
 - Signal
 - Shortcomings
- Exceptions in C++
 - Exception Scope (try)
 - Exception Arguments (catch)
 - Exception Matching
 - Exception Raise (throw)
 - Advantages

This is a module outline as you know the lower part; the blue part is what you will be discussing in the module today.

(Refer Slide Time: 01:37).

The slide has a dark blue header with the title 'Expectations'. On the left, there's a sidebar with a logo, the text 'Module 37', 'Partha Pratim Das', 'Objective & Outline', and a list of topics under 'Exceptions in C++'. A small circular video player shows a man speaking. The main content area contains a bulleted list of expectations:

- Separate Error-Handling code from Ordinary code
- Language Mechanism rather than of the Library
- Compiler for Tracking Automatic Variables
- Schemes for Destruction of Dynamic Memory
- Less Overhead for the Designer
- Exception Propagation from the deepest of levels
- Various Exceptions handled by a single Handler

At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

As we move to C++, certainly the designers had been very sensitive and cautious about incorporating the error handling as the part of the language. The first expectation is we should be able to separate error handling code from ordinary code. Ordinary code means a normal flow. The normal flow and the error handling as we saw are often intermixed we would try to separate this out and that is a requirement of exception handling in C++.

Second it is a language mechanism rather than of the library, that is the language should at the core take cognizance of the fact that errors can happen exceptions will need to be managed and therefore provide a language feature for doing that.

Given as a language feature compilers would be able to track all automatic variables and take care of the lifetime destroying them when there is some situation to terminate or if not terminate a whole program, but to abnormally terminate the execution of a certain function the automatic variables will be taken care off. There are schemes for destruction of a dynamic memory and we will certainly aspect that the overhead for the designers which we saw in the C style would should be much less.

This is very very important which we look at is often the exception does not happen in the main, it will happen in function which is in a deep nested call; main has called one function; that has called another function; that has called yet another function and so on. Then in some 3, 4 levels, 5 levels of calls or 10 levels of calls some exception has happened so you need to come out of this deep nesting of calls in the exception situation so that should be possible. And variety of situations, we should be able to handle through a single handler.

(Refer Slide Time: 03:39)

```

Module 37
Partha Pratim Das
Objective & Outcome
Exceptions in C++
Exception Scope (try)
Exception Arguments (throw)
Exception Matching
Exception Rethrow (throw again)
Summary

try-throw-catch

void f()
{
    A a;
    try {
        B b;
        a();
        h();
    }
    catch (UserException ex) {
        cout << ex.what();
    }
    return;
}

class UserException : public exceptions {
public:
    void g()
    {
        A a;
        UserException ex("From g()");
        throw ex;
    }
    return;
}

• g() called and exception raised
• Exception caught by catch clause

```

These are the expectation with which the exception mechanism in C++ has been designed and this is a highest show the same illustration which I was showing earlier in the earlier module with the non local goto. So there is a function f which is calling a function g and this is function g. Now what it does, it basically does something called a throw, we will see what does throw mean. And this is what some kind of what will say is a exception class.

Once g is called then certainly the first situation is a happy part, the normal flow; so g will come out through the return. If g comes our through the return it has a normal flow the control goes to the next statement h which is clear. But if g is called and because of

something happening in between because of some error situation arising in g it come across an error situation that will throw an exception.

Which means it will throw is a keyword, it will do throw and then it will put an object of the exception class; any class can be used as an exception class really. And if it throws then the control does not come back to h, does not come back to the statement immediately following g it comes back to what is known as a catch clause. So from here the control will come out here and then you can take care of the exception at this point. So it is caught by exception clause.

If you look into what all have become different is now we have a try; we say it is a try block, try is a keyword, catch is another keyword, so there is a catch block and throw is another keyword. We will say try throw catch; kind of or try catch for simple terms. So what we do in the caller, we put the calls within a try block and a try block has an associated catch. And the call function may throw saying that the exception has happened.

When it throws the basic behaviour is at this point the function was called if it throws the basic is the try does not continue, the control comes back to the try, but it does not continue in the immediately next statement, but it goes to the catch clause. In the catch clause it tries to find what kind of exception it as got and accordingly it will try to execute the catch clause. So it will execute this catch clause and then it will continue here.

Basically, there is some situation that has happened here which is given rise to the function g realising that exceptions needs to be reported, error needs to be reported. The error object is created here, then it is reported here I am talking about the exception stages, this is a create stage if I cleanly draw it. This is a create stage where you create that this is a report stage.

And the interesting thing is we do not need to do anything special for the detect because the catch clause as is present will detect it immediately then it takes care of this in the code this is called the handler code, the catch handler or the exception handler, handles it

and then you come to the next where you have the recover. This is how the exception will get handled in terms of a C++ mechanism.

(Refer Slide Time: 07:19)

The slide title is "Exception Flow". The left sidebar shows "Module 37" and "Partha Pratim Das". The main content area contains C++ code demonstrating exception handling:

```
#include <iostream>
#include <exception>
using namespace std;

class MyException : public exception {};
class MyClass {};

void h0 (MyClass a) {
    //throw 1;
    //throw 2.6;
    throw MyException();
    //throw exception();
    //throw MyClass();
}

void g0 (MyClass a) {
    try {
        h0();
    }
    catch (int) { cout << "int"; }
    catch (double) { cout << "double"; }
    catch (...) { throw; }
}

void f0 () {
    try {
        g0();
    }
    catch (MyException) { cout << "MyException"; }
    catch (exception) { cout << "exception"; }
    catch (...) { throw; }
}

int main() {
    try {
        f0();
    }
    catch (...) { cout << "Unknown"; }
    return 0;
}
```

A hand-drawn diagram on the right illustrates the call flow: "main" calls "f0", "f0" calls "g0", and "g0" calls "h0". Red annotations include "Call all" pointing to the catch block in f0, and "Catch all" pointing to the catch block in g0.

Let us look at a more detailed example and try to understand this. I have couple of slides following this where all these mechanisms are explained, but first I would explain it through an example. So, let us first look at what the example has, forgetting about the exception situation there is a function h, there is a function g, there is a function f and main function. And it is a simple nested call main calls f, f in turn calls g, g in turn calls h, h does something. These are basic structure. The difference that we are doing is when main calls f it puts it within a try block and has an associated catch.

When f calls g puts it in a try block, has multiple associated catch blocks. When g calls h it puts in a try block has multiple associated catch clause and h does the task, this is the situation. If we look into any of g, f, or main, we see that they are calling the function within the try block expecting or rather apprehending that this function might fail and if it fails it will throw something and whatever it throws I will catch in the associated catch clause.

Now let us see why it is so many catch clauses and what happens with that and so on. Let say I have commented here all of them because certainly if one throw happens in the function is gone. Suppose this is not commented, so what happens is it is doing something comes across an error and it throws one. If it throws one what does that mean? What is one? Basically it is throwing an int kind of object. If it throws then the controls goes, if we look into the calls stack then what do we have we have main, main has called f, f has called g, g has called h. This is where we are at present.

Now if you do throw, then it has detected that it cannot proceed any further, so throw is pretty much like as if you want in terms of control flow you are trying to go out of this function. So you want to go out and go back to g. Now why do you go back in g? In g the call of h is guarded within the try block, so it does not go back to this point. Rather as you throw the control goes to the list of catch clauses that you have. And what have been thrown is one, which is an int.

Now once you get there what happens is something very very interesting it takes this object, object one and goes over this list of catch clauses and tries to match. If it finds a match then it immediately stops there and executes the corresponding handler. If it does not find a match it proceeds further. So what happens in case of if one has been thrown it is an int object, so catch is expecting an int since it is expecting an int it matches and it will execute this cout int. This is just saying that it is as if this is a handler.

Once this is done, then the control will jump the remaining of the catch clauses and some here and continue. Now, whatever g has to do; now the whole thing has already taken place, the error happened, it was reported, it has been detected here, it has been handled here, you recovered and you are proceeding to this part.

Let us see what happens if let us say again draw the stack that will be important in every case h, g, f main. Now let us say this is where something was happening and it reaches a point where it throws 2.5, just to show that that is a something different has happened.

Now what is 2.5? 2.5 is an object of the double time we know. As it throws the control comes back to g. Where does it come back? It does not come back here; it comes back to

the beginning of the catch clause, because something has been thrown. And you starts matching it has a double object it tries to match int. it will fail if it will not be able to match. Catch clause is do not match by implicit conversion.

If we look into this match strategy this looks pretty much like the overload matching. So as if you can think of then thrown as if is calling a function with a parameter one single parameter. And there are several functions as if here, three functions here which takes an int x a double and takes some dot dot dot and you are trying to resolve which particular function, which particular catch clause will be resolved.

But there are two major distinctions from the overload resolution; one is overload resolution allows you for implicit conversion. For example int will be matched with double double will be truncated and matched within and so on. This is not allowed in deciding on the catch clause. The second is overload resolution happens over the set of functions as a whole. Here it will happen according to this order. And as soon as you found a match you will not try to find if there is a better match, you will just end there and call that catch clause. So what will happen here? They have a double object; this will not match, but this will match. Now, it will do print double and then it will recovered continues here that is a sample thing.

Let us continue on this and try to see what happens next. Main, f, g, h, let us check this. Here what is my exception, there is a standard library exception which has a class called exception predefined which takes care of all different kinds of exception that can happen in C++. I have defined a class which is a specialization of exception class, so some throwing an object of my exception. What happens again? As the control comes to g that is starts when this tries to match with int does not match have a my exception object, tries to match with catch the double does not match, then tries to match here, what is this? This is known as ellipses.

Ellipses are three dots they are supposed to match with anything. Whatever you have that will match with that. So, ellipses will match with one it will match with double, it will match with my exception, it will match with exception anybody, but earlier cases when we are thrown 1 or we had throw 2.5 since int and double catch clauses where before this

ellipses then it did not fall through to this point. But here with my exception these two did not match so it comes here and matches at this point this is also known as a catch all clause because it matches everything.

So it goes to handle that, and what is a handle do? Handle does something very simple it says throw which means that it is not going to do anything but as the h function has thrown a my exception object this will simply re throw that object to the caller. So which means it will re throw it will go back go upwards to its caller it will the stack will come down to its caller. The caller was f, caller had called g and you have thrown, what is your exception object? Exception object is what you had received from h, my exception object.

So since you have come here through an exception being thrown you will continue on the catch clauses. So, now you have a my exception object and you have a my exception catch clause so it matches say my exception skip over the other catch clauses and then you go to the remaining of the f function and continue. So, it is not necessarily that the caller will be able to handle the exception that you throw, the caller might handle that, caller might propagate it higher up to for the caller to handle that and we say that is re throwing an exception this is what you have done specially here.

Let us move forward and see what if I throw an exception of the class exception of which my exception was a specialization. Now, what will happen is, I come again here it goes to this point because these two do not match I throw, I come, I come here, match the exception, I can see the exception. Suppose I throw, my class is some other class and I throw an exception of my class it will start looking here it will go throw the catch clauses, finally it matches at this point it throws again goes through this, this, none of them match my class so it thrown again, so it comes back to the try block in main goes to this catch clause where it matches again and you throw and you are saying that something unknown as happened. This is how the exceptions will get propagated from a deep function to the outer called function.

And there are some interesting things that you can note, for example, suppose I did not have this catch clause. And say I have thrown my exception then what will happen, this

does not match, this does not match. Now there is no other catch clause left so it falls here, now the default behaviour of the exception mechanism is if a caller has received an exception which it has not been able to handle, because there was no catch clause, it by default will re throw it, and it will by default will re throw the same exception. So even without this we will get the same behaviour that it will go back and go to that my exception.

Now, if that is a case then why do we really have re throw, the reason we have re throw is in many cases I may have some catch condition and I may first want to handle something which I am capable off, and then I may want that my caller should also know it. So I may want to handle some and then again throw it because if I just handle it like I did here then the system understands that the exception is taken care of and it will proceed right here and it will not go to the caller it will not throw. But if I handle and then throw again, then it will take that a fresh exception has happened from the function g and we will go to its caller f. So this is the basic mechanism of exception that happens.

Now let me also show you what happens in terms of the stack, say main, f, g, h, let us say my exception has happened. Now it throws which means that the control has to come out of this function g, comes out of it. So if it comes out of that which means that the stack frame which was associated with this function g will have to be removed. When this as to be removed what happens to this local object, this local object is been created. So, what exception does is normally what would have happened is at the normal exist point which is here the destructor of this function that is a colon tilde my class would have got invoked.

Now, you are existing from this point, now you are coming out from this point because you have done a throw. But even at this point the destructor of this local object will get called. So all destructors of all object which are live at this point then destructors will get called. And only after that the stack frame will be squashed and the control will go back to g.

Now if we have thrown my exception then certainly it does not match here, does not match here, it matches here, and it re throws. So the control goes back from g to f which

means that the stack frame of g will also have to be unwound and that will be done at this point, that will be done at this point, and this local object of g will get destroyed. So the basic advantage of exception base flow is that it not only takes care of your automatically going up in the stack based on the call sequence and the try blocks and matches the corresponding catch clause in the try block. Whenever it goes out of a function because the function has thrown it destroys all the local objects that were present.

At any point of time, if you have a thrown from here it has been passed on throw this and now you are working on f, you know that for g and h; the two functions that have thrown all local objects have been properly cleaned up. The stack has been as it is called this process of removing, destroying the local objects automatic objects and removing the stack frames as you throw objects from one function to its parent is known as stack unwinding the stack is collapsing and till you get to the function where you are supposed to continue. This is the basic exception mechanism which is based on the try block.

Certainly, let me also mentioned another point. let say suppose this was there and let say in a function g instead of this you have just written g. Suppose you have called a function which is not within the try block and then this my exception has been raised. If my exception as been raised does not match here, does not match here, from this re throw will happen, so my exception happens here. After this g may be there is some other statement.

Now what happens is, if you this is not within the try block also, then if you get an exception in g then once the control returns because of that exception you do not continue at the next statement, you immediately re throw and raise the same exception object to the parent. These are default mechanism. This default mechanism is important because if you forget to put try block around some functions calls which may have some exceptions raised then you will have a test of a difficulty of not being able to handle the error, so by default if you have not raised if you have not put a function within a try block then you simply pass it on to a parent, your parent keeps on doing that.

Finally, if you have some exception in the main which is not guarded by a try block or there is a catch clause. Suppose, here I did not have this catch clause, suppose here I had some catch clause for some other class and so on. But I did not have a catch all clause then it is quite possible that an exception like thrown like my class object will not be caught at this point because it does not match here. In that case the main cannot certainly throw, because it does not have a caller. So in that case a termination handler will get called. So, that is a basic mechanism of exceptions that happen.

(Refer Slide Time: 26:11)

try Block: Exception Scope

- try block
 - Consolidate areas that might throw exceptions
- function try block
 - Area for detection is the entire function body
- Nested try block
 - Semantically equivalent to nested function calls

<p>Function try</p> <pre>void f() { try { throw E(); } catch (E& e) { } }</pre>	<p>Nested try</p> <pre>try { try { throw E(); } catch (E& e) { } } catch (E& e1) { }</pre>
--	---

NPTEL MOOCs Programming in C++ Partha Pratim Das 9

What we have done in the next couple of slides, whatever I have discussed those are documented here for example with some more details. For example, a function as a whole may have a try block. So, all that you do is instead of putting multiple try block you say that a function and the body of function header and this is the function body. So, in between that you write try so which means that the whole functions is a part of try. Naturally the try blocks can be nested within try block you could have other blocks and so on, but try block must have a catch block with that.

(Refer Slide Time: 26:47)

The slide title is 'try-throw-catch'. On the left, there's a sidebar with a logo and a list of topics: Module 37, Partha Pratim Das, Objective & Outline, Exceptions in C++, Exception Scope (try), Exception Arguments (catch), Exception Matching, Exception Return, Details, Advantages, Summary. Below the sidebar is a video player showing a man speaking.

The main content area contains two code snippets:

```
void f() {
    try {
        b();
        g();
        h();
    } catch (UserException ex) {
        cout << ex.what() << endl;
    }
    return;
}
```

```
class UserException : public exception {
public:
    void g() {
        UserException ex("From g()");
        throw ex;
    }
    ~UserException() {}
};
```

A bullet point below the first code snippet says '• try Block'.

At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

This is from the earlier example just highlighting what is that.

(Refer Slide Time: 26:50)

The slide title is 'catch Block: Exception Arguments'. On the left, there's a sidebar with a logo and a list of topics: Module 37, Partha Pratim Das, Objective & Outline, Exceptions in C++, Exception Scope (try), Exception Arguments (catch), Exception Matching, Exception Return, Details, Advantages, Summary. Below the sidebar is a video player showing a man speaking.

The main content area has a bullet point '• catch block' followed by a list:

- Name for the Exception Handler
- Catching an Exception is like invoking a function
- Immediately follows the try block
- Unique Formal Parameter for each Handler
- Can be simply a Type Name to distinguish its Handler from others

At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

Then catch block has a different arguments that we have seen every catch block will have one argument and that has to be unique between the catch blocks.

(Refer Slide Time: 27:03)

The slide title is "try-throw-catch". It features a sidebar with a navigation menu and a video player showing a professor. The main content area contains two code snippets. The first snippet, labeled "void f()", shows a try block with two catch blocks. The second snippet, labeled "class UserException", shows a throw expression. A bullet point at the bottom left says "• throw Expression".

```
void f() {
    try {
        b();
        g();
        h();
    } catch (UserException ex) {
        cout << ex.what();
    }
}

class UserException : public exception {
public:
    void g() {
        K *at;
        UserException ex("From g()");
        throw ex;
    }
}
```

- throw Expression

So, the arguments here have to be unique. And there are two interesting things to note one, is how you should pass this argument, should you pass it as a value or as a reference. Usually it is passed as a reference. Particularly if it is a user defined object type the reason you would like to pass it as a reference because then you do not need to copy that exception object you can just keep on propagating.

And whenever it the handler ends, whenever the handler terminates if it has not re thrown whenever the handler terminates it will destroy that exception object. It is also interesting that the exceptions object are kind of so exception objects becomes kind of automatic objects because they are constructed at the point when they are thrown or before that in the function that it is throwing them and they will get destructed in some other function scope in the catch handler. There is kind of a very interesting automatic object which has a lifetime scope which is dependent on the runtime not necessarily on the compile time which is typical for all other automatic objects.

And therefore, the exception objects will always have to be created in the free store in the stack not in the stack, but in the, what you say as a heap. Because you cannot it in the stack because you do not know for how long you will need to maintain this object. So, you do not know whether the stack frame of a function in which you create that how long

that stack frame will exist. So you will typically create these objects are created in the free store.

(Refer Slide Time: 28:46)

The slide has a blue header with the title 'try-catch: Exception Matching'. On the left, there is a sidebar with a logo, the text 'Module 37', 'Partha Pratim Das', 'Objective & Outline', and a list of topics including 'Exception in C++', 'Exception Scope (try)', 'Exception Arguments (catch)', 'Exception Matching', 'Exception Name (throw)', 'Advantages', and 'Summary'. Below the sidebar is a video player showing a man speaking. The main content area contains three bullet points: 'Exact Match' (with sub-points about type matching and no implicit conversion), 'Generalization / Specialization' (with a point about public base class matching), and 'Pointer' (with a point about pointer types being convertible by standard conversion).

We have talked about exception matching.

(Refer Slide Time: 28:51)

The slide has a blue header with the title 'try-throw-catch'. On the left, there is a sidebar with a logo, the text 'Module 37', 'Partha Pratim Das', 'Objective & Outline', and a list of topics including 'Exception in C++', 'Exception Scope (try)', 'Exception Arguments (catch)', 'Exception Matching', 'Exception Name (throw)', 'Advantages', and 'Summary'. Below the sidebar is a video player showing a man speaking. The main content area shows two code snippets. The first snippet is a 'f()' function:

```
void f() {
    A a;
    try {
        B b;
        g();
        h();
    } catch (UserExcept ex) {
        ex.what();
    }
    return;
}
```

The second snippet is a 'g()' function:

```
class UserExcept : public exceptions {
public:
    void g() {
        A a;
        UserExcept exFrom(g());
        throw ex;
    }
    ...
}
```

Below the code, there is a bullet point: '• Expression Matching'.

This is just detailed here is to what the exception that you throw, how that is matched here. You will have to remember that if you have an exception and its specialization then certainly when you write the clauses you should first write the catch clause for the base class and then the catch clause for the specialised class. Because you remember the exceptions are catch is matched from one end to the other. If you have a specialised class exception thrown then it will then I saying the right thing, I am sorry I am saying the wrong thing. So, if I have exception here, and my exception here, then I have two catch clauses; one for my exception and another for the exception.

Now, what I am trying to point out is if you have an exception of the base class then certainly it will not match here, because this means a down casting but it will go up and match there. So, instead of this, if you had say it has my exception here then you will have a problem because when you have an exception of the derived class then it will match here because you are doing a up cast.

So, you will have to remember that whenever you have a hierarchy of classes being used for the exception then the specialised one has to come higher earlier and the generalised one have to come later. So that mechanism will have to be followed.

(Refer Slide Time: 30:46)

The slide has a blue header with the title 'try-catch: Exception Matching'. On the left, there's a sidebar with a logo, the text 'Module 37', 'Partha Pratim Das', and a list of topics: 'Objective & Outline', 'Exceptions in C++', 'Exception Scope (try)', 'Exception Arguments (except)', 'Exception Matching (Base Class)', 'Advantages', and 'Summary'. Below the sidebar is a circular video player showing a person speaking. The main content area contains a bulleted list of rules for exception matching:

- In the order of appearance with matching
- If Base Class catch block precedes Derived Class catch block
 - Compiler issues a warning and continues
 - Unreachable code (derived class handler) ignored
- catch(...) block must be the last catch block because it catches all exceptions
- If no matching Handler is found in the current scope, the search continues to find a matching handler in a dynamically surrounding try block
 - Stack Unwinds
- If eventually no handler is found, terminate() is called

At the bottom of the slide, there are footer links: 'NPTEL MOOCs Programming in C++', 'Partha Pratim Das', and '15'.

This is all described in this point so you can go throw that.

(Refer Slide Time: 30:50)

The slide has a blue header with the title 'throw Expression: Exception Raise'. On the left, there is a sidebar with a logo, the text 'Module 37', 'Partha Pratim Das', and a navigation menu including 'Objective & Outline', 'Exceptions in C++', 'Exception Scope (try)', 'Exception Arguments (catch)', 'Exception Matching', 'Exception Raise (throw)', 'Advantages', and 'Summary'. A small video thumbnail of a man speaking is also present. The main content area contains a bulleted list:

- Expression is treated the same way as
 - A function argument in a call or the operand of a return statement
- Exception Context
 - class Exception ;
- The Expression
 - Generate an Exception object to throw
 - throw Exception();
 - Or, Copies an existing Exception object to throw
 - Exception ex;
 - ...
 - throw ex; // Exception(ex);
- Exception object is created on the Free Store

At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

We have talked about raising the exception, what happens in throw.

(Refer Slide Time: 30:57)

The slide has a blue header with the title 'throw Expression: Restrictions'. On the left, there is a sidebar with a logo, the text 'Module 37', 'Partha Pratim Das', and a navigation menu including 'Objective & Outline', 'Exceptions in C++', 'Exception Scope (try)', 'Exception Arguments (catch)', 'Exception Matching', 'Exception Raise (throw)', 'Advantages', and 'Summary'. A small video thumbnail of a man speaking is also present. The main content area contains a bulleted list:

- For a UDT Expression
 - Copy Constructor and Destructor should be supported
- The type of Expression cannot be
 - An incomplete type (like void, array of unknown size or of elements of incomplete type, Declared but not Defined struct / union / enum / class Objects or Pointers to such Objects)
 - A pointer to an incomplete type, except void*, const void*, volatile void*, const volatile void*

At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

This is the exception showing that.

(Refer Slide Time: 31:00)

The slide title is '(re)-throw: Throwing Again?'. It features a sidebar with a logo, the name 'Partha Pratim Das', and a navigation menu for 'Module 37' including 'Exception in C++', 'Exception Scope (try)', 'Exception Arguments (const)', 'Exception Handling', 'Exception Raise (throw)', 'Advantages', and 'Summary'. A video thumbnail of the speaker is also present.

• Re-throw

- catch may pass on the exception after handling
- Re-throw is not same as throwing again!

Throws again	Re-throw
try { ... } catch (Exception& ex) { // Handle and ... // Raise again throw ex; // ex copied // ex destructed }	try { ... } catch (Exception& ex) { // Handle and ... // Pass-on throw; // No copy // No Destruction }

NPTEL MOOCs Programming in C++ Partha Pratim Das 19

All these we have discussed now, the re throw what happens in terms of re throw that we have discussed.

(Refer Slide Time: 31:10)

The slide title is 'Advantages'. It features a sidebar with a logo, the name 'Partha Pratim Das', and a navigation menu for 'Module 37' including 'Exception in C++', 'Exception Scope (try)', 'Exception Arguments (const)', 'Exception Handling', 'Exception Raise (throw)', 'Advantages', and 'Summary'. A video thumbnail of the speaker is also present.

- **Destructor-savvy:**
 - Stack unwinds; Orderly destruction of Local-objects
- **Unobtrusive:**
 - Exception Handling is implicit and automatic
 - No clutter of error checks
- **Precise:**
 - Exception Object Type designed using semantics
- **Native and Standard:**
 - EH is part of the C++ language
 - EH is available in all standard C++ compilers

NPTEL MOOCs Programming in C++ Partha Pratim Das 20

Finally, before we conclude you would like to note that there are several advantages of this try catch for mechanism. It is a destructor savvy because it take and on mind the

stack and clean up the local object is a non obtrusive because it separates out the whole code clutter into separate normal flow and the exception flow it is precise. It is native and standard, in the sense is the part of the language not part of a third party or stand standard library like that.

(Refer Slide Time: 31:42)

The screenshot shows a presentation slide with a blue header bar containing the title 'Advantages'. On the left side, there is a sidebar with a logo at the top, followed by 'Module 37' and 'Partha Pratim Das'. Below this, a list of topics is shown: 'Exception in C++', 'Exception Scope (try)', 'Exception Arguments (catch)', 'Exception Handling', 'Exception Base Classes', 'Advantages', and 'Summary'. A circular video player window in the bottom-left corner shows a man speaking. The main content area contains two bullet-point sections: '**Scalable:**' and '**Fault-tolerant:**'. The 'Scalable' section lists three points: 'Each function can have multiple try blocks', 'Each try block can have a single Handler or a group of Handlers', and 'Each Handler can catch a single type, a group of types, or all types'. The 'Fault-tolerant' section lists three points: 'Functions can specify the exception types to throw; Handlers can specify the exception types to catch', 'Violation behavior of these specifications is predictable and user-configurable', and 'The exception handling mechanism is highly efficient and provides a clean way to handle errors'. At the bottom of the slide, there is footer text: 'NPTEL MOOCs Programming in C++', 'Partha Pratim Das', and a page number '28'.

In that process this is quite scalable and quite fault tolerant. Scalable, in it does not matter as to how many try blocks you have, how much of nesting you have, how much catch clauses you have, this will work in every case in a right manner.

(Refer Slide Time: 31:59)

The slide is titled "Module Summary" and is part of "Module 37" by Partha Pratim Das. The slide content includes a bulleted summary of the module's objectives:

- Discussed exception (error) handling in C++
- Illustrated try-throw-catch feature in C++ for handling errors
- Demonstrated with examples

On the left sidebar, there is a vertical navigation menu with the following items:

- Partha Pratim Das
- Objective & Outline
- Exceptions in C++
 - Exception Base (try)
 - Exception Objects (catch)
 - Exception Matching
 - Exception Rethrow
 - Advantages
- Summary

At the bottom of the slide, there is a small video thumbnail showing a man speaking, the text "NPTEL MOOCs Programming in C++", the name "Partha Pratim Das", and the number "22".

With this to summarise we have discussed the exception or error handling in C++ particularly we have try to understand the try, catch, throw mechanism of C++ which takes care of all possible ways the exceptions can happen and all possible ways you can handle that.

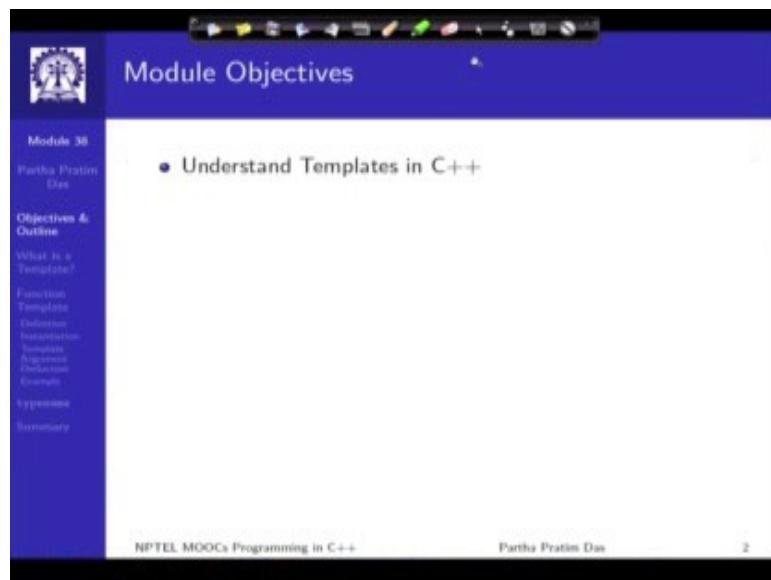
Actually if you can continue to design properly with try throw catch in C++ you would not need any of the exception mechanism in C. Of course you will still need to use them because some of the system calls do use those error mechanisms like, putting signals or putting error number. So you will still need to use for those cases, but for your own code you will never need to use any of the C error handling or C standard library functions.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 54
Template (Function Template): Part 1

Welcome to module 38 of programming in C++. We are almost at the end of the course and in this module and the next we introduce very different concepts in C++ programming, which is known as generic programming concepts. And it is also referred to as meta programming concepts. This is done through a feature called templates. We will go through what templates are and why they are required, but the core idea of template is to be able to write a code, which at the time of compilation can generates further the new code and that generated code then gets complied again. So, templates is not only a program, but it is a meta program which generates other programs

(Refer Slide Time: 01:26)



So, we will cover this in two modules, naturally the objective will be to understand templates.

(Refer Slide Time: 01:33)

Module Outline

- What is a Template?
- Function Template
 - Function Template Definition
 - Instantiation
 - Template Argument Deduction
 - Example
- typename
- Class Template
 - Class Template Definition
 - Instantiation
 - Partial Template Instantiation & Default Template Parameters
 - Inheritance

NPTEL MOOCs Programming in C++ Partha Pratim Das

And this is what are outline would be the blue part is what we will do first which will significantly covered the function template. And in the next module, we will talk about the class template.

(Refer Slide Time: 01:48)

- Function search, min etc.
- The basic algorithms in these functions are the same independent of types
- Yet, we need to write different versions of these functions for strong type checking in C++
- Classes list, queue etc.
- The data members and the methods are almost the same for list of numbers, list of objects
- Yet, we need to define different classes
. A small video thumbnail of Partha Pratim Das is in the bottom right."/>

What is a Template?

- Templates are specifications of a collection of functions or classes which are parameterized by types
- Examples:
 - Function search, min etc.
 - The basic algorithms in these functions are the same independent of types
 - Yet, we need to write different versions of these functions for strong type checking in C++
 - Classes list, queue etc.
 - The data members and the methods are almost the same for list of numbers, list of objects
 - Yet, we need to define different classes

NPTEL MOOCs Programming in C++ Partha Pratim Das

Now, what is Template? Templates are specifications of a collection of a function or

classes which are parameterized by types, this is the important part. They are parameterized by type which is a complete a new concept now coming in. In that so far we always knew that any code that we write in C++ must have a fixed type is a strongly type language whether it is int or it is double, or it is char star, or it is a user-defined type, it must be known. But here we are now seeing that the type can be also a parameter and the motivation comes from the fact that we want to increase the code reused in C++.

There are several algorithms which are generic in terms of algorithmic strategy like finding maximum or minimum of two numbers or searching a set of data items which can be compared, but depending on the specific type their codes become different. Similar things happen with a lot of class designs if we talk about stack all that we know is a LIFO structure, last in first out infrastructure, but depending on the specific type of elements that we have to deal with the stacks using a character elements, a stack using an integer element, a stack using some user defined type elements will all be different. So, template is an attempt in a trying to combine all this into a single binding code.

(Refer Slide Time: 03:25)

Function Template:
Code reuse in Algorithms

Module 38
Partha Pratim Das
Objectives & Outline
What is a Template?
Function Template
Inductive Induction
Template Argument Substitution
Example - Hypothesis
Summary

- We need to compute the maximum of two values that can be of:
 - int
 - double
 - char * (C-String)
 - Complex (user-defined class for complex numbers)
 - ...
- We can do this with overloaded Max functions:


```
int Max(int x, int y);
double Max(double x, double y);
char *Max(char *x, char *y);
Complex Max(Complex x, Complex y);
```

With every new type, we need to add an overloaded function in the library!
- Issues in Max function
 - Same algorithm (compare two value using the appropriate operator of the type and return the larger value)
 - Different code versions of these functions for strong type definition in C++

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, let us just understand this little bit in more depth by taking examples. Suppose, I want to compute a maximum of two values, and the values could be integers value, there could be double values they could be strings, so in C string, they are they are char star

they could be complex numbers and so on they could be so many others. Now, if I want to do a max will have a signature of max, I have one value, I have other value, I have the results. So, this is a signature of max which is kind of would come in something like this. If it is doing it for int, if I am doing it for double it will look like this for strings for complex and so on. So, certainly we know that unlike in C were writing such functions were really difficult because you could not reuse the function name for different types C++ allow us to overloads. So, with the overloading, we can actually write a number of overloaded functions all of which will be max and we can just use that.

Now, if we look into these overloaded functions then we see few we observe a few things; one is they have the same algorithm the way to find out max is to do compare two values find out which one is larger and then returns that larger value that is the basic you know algorithm for. So, the algorithm is same in respective of which overload which we use, but since C++ is strongly typed for every version, we need to have a separate code we need to have a separate version of code which will work the code for int will not work for double and so on. So, that is a lot of you know code duplications that need to be done. So, that is basic motivation for looking at some solution to optimize all this.

(Refer Slide Time: 05:19)

```

#include <iostream>
#include <cstring>
#include <cmath>
using namespace std;

// Overloads
int Max(int x, int y) { return x > y ? x : y; }
double Max(double x, double y) { return x > y ? x : y; }
char *Max(char *x, char *y) { return strcmp(x, y) > 0 ? x : y; }

int main() {
    int a = 3, b = 6, dMax;
    double c = 2.1, d = 3.7, dMax;

    cout << "Max(" << a << ", " << b << ")" << endl;
    cout << "Max(" << c << ", " << d << ")" << endl;

    char *a1 = new char[6], *a2 = new char[6];
    strcpy(a1, "black"); strcpy(a2, "white");
    cout << "Max(" << a1 << ", " << a2 << ")" << endl;
    strcpy(a1, "white"); strcpy(a2, "black");
    cout << "Max(" << a1 << ", " << a2 << ")" << endl;

    return 0;
}



- Overloaded solutions work
- In some cases (C-string), similar algorithms have exceptions
- With every new type, a new overloaded Max is needed
- Can we make Max generic and make a library to work with future types?
- How about macros?

```

So, first is if we will look at max as overload. So, these are the different overloads. You

can see that this is overload for int, this for double, this is for char star, c string now naturally if you c strings are represented by pointer to character. And in that you are not really interested to compare the pointers, but you want to compare the strings. So, here your algorithm is little bit different, but it falls in the same category or structure. And with that, you can print the max value of a different integers pairs, double pairs, string pairs and so on.

So, this will work, but the catch in this as the problem here is the solution works and in some cases exceptional I mean different kind of code need to be written, but the main problem is, if I have a new type, I will need to write the max all over again, I have to overload and write that. So, every time I add a new type to my system if it can be compared and I want to do a max, I have to write a separate code. So, the point is can we make the reuse stronger can we make max generic, so that it not only will work with the currently known types it will also work for the types that will be defined in future. So, that is an interesting problem. So, can we do that for, you will immediately come and say that is possible you in C already that is called macros.

(Refer Slide Time: 06:50)

```
#include <iostream>
using namespace std;

#define Max(x, y) ((x) > (y)) ? x : y

int main() {
    int a = 3, b = 5;
    double c = 2.1, d = 3.7;

    cout << "Max(" << a << ", " << b << ") = " << Max(a, b) << endl;
    // Output: Max(3, 5) = 5

    cout << "Max(" << c << ", " << d << ") = " << Max(c, d) << endl;
    // Output: Max(2.1, 3.7) = 3.7
}

return 0;
}

• Max, being a macro, is type oblivious – can be used for int as well as double, etc.
• Note the parentheses around parameters to protect precedence
• Note the parentheses around the whole expression to protect precedence
• Looks like a function – but does not behave as such
```

So, let's look back into macros. So this is a macros, this is a macros. I can write the max function in terms of a macro like this and if I do this for int double. I do not really

when I write the macro, I do not need really need to know what the type is whether it is int or it is double. I can simply write this because it gets replace at the point, where the macro is instantiated and therefore, the corresponding expression works. So, this is this is fine it looks like a function, but it does not really behave like a function. But just you remind you that the moment you write macros your into lot of lot of different problems, one a certainly is the way you write the macro you see that the parameters are parenthesized here.

The reason is unless you do that then x itself could be an expression which will get replaced here and that if that expression has operators which have a precedence, which is lower than this greater then operator then you will have a whole of this expression mixed up. Similarly the whole expression needs a parenthesis around it because otherwise for example, if you want to use it here as I did. So, here this whole expression is coming up if you do not have these then the precedence of this question mark colon and the precedence of the output indirection operators are in the wrong order. So, it tries to output only this part and then put a question mark column on the top of that. So, these are these are some of nuances you have live with, but then it look likes, can we can we live this solution.

(Refer Slide Time: 08:21)

The slide has a blue header with the title 'Max as a Macro: Pitfalls'. The left sidebar contains a navigation menu with items: Module 38, Partha Pratim Das, Objectives & Outline, What is a Template?, Function Template, Definitions, Instantiations, Assertions, Deduction, Examples, Hypotheses, Summary. The main content area shows a C++ code snippet with annotations explaining common pitfalls:

```
#include <iostream>
#include <cstring>
using namespace std;

#define Max(x, y) ((x) > (y))? x: y

int main() {
    int a = 3, b = 6;
    double c = 2.1, d = 3.7;

    // Side Effects
    cout << "Max(" << a << ", " << b << ") = " // Output: Max(3, 6) = 6
    cout << Max(a+, b++) << endl;
    cout << "a = " << a << ", b = " << b << endl; // Output: a = 4, b = 7

    // C-String Comparison
    char *a1 = new char[6], *a2 = new char[6];
    strcpy(a1, "black"); strcpy(a2, "white");
    cout << "Max(" << a1 << ", " << a2 << ") = " << Max(a1, a2) << endl;
    // Output: Max(black, white) = white

    strcpy(a1, "white"); strcpy(a2, "black");
    cout << "Max(" << a1 << ", " << a2 << ") = " << Max(a1, a2) << endl;
    // Output: Max(white, black) = black

    return 0;
}
```

- ★ In "Side Effects" – the result is wrong, the larger values gets incremented twice
- ★ In "C-String Comparison" – swapping parameters changes the result – actually compares pointers

NPTEL MOOCs Programming in C++ Partha Pratim Das 8

Now, so I would remind you to some of the pitfalls this is one. So, I would like to given a, and b, I would like write max a ++, b ++, thinking that this will give me the maximum value from the original value if a, and b and then it will give me the incremented value. So, the original values being 3 and 5, I am expecting an output 5 naturally, because of the original values. But if you actually work this out, you will find that the output of max is given as 6.

And after this is completed, the incremented values, the first value is incremented ones, the second will be incremented twice or basically the larger value gets incremented twice, because it turns out to be the expression is basically a ++, b ++ question mark, a ++ ,b ++, this is what you actually got. So, depending on whichever was larger possibly in this case, this was larger. So, b plus plus got incremented twice. So, this is that is kind of a, so this is quite counter intuitive to this.

Then you have issue of if you have two character pointers in want to compare them, and you will simply use one pointer keeps use a string black, other gives a string white. Certainly, in this part of the code, if you look carefully all the time doing is I have just stopped the order in which they are invoked; s1 and s2, and the result become different. The reason is simply, because here when we write the macro, it basically compares the value of the pointers not the string that are pointed to by those pointers. So, if we change that order this will be basically change.

(Refer Slide Time: 10:11)

The screenshot shows a presentation slide titled "Function Template". The slide content is as follows:

Module 3B

Partha Pratim Das

Objectives & Outline

What is a Template?

Function Template

Definitions
Instantiation
Template
Definition
Example
Typeoyer
Summary

• A function template

- describes how a function should be built
- supplies the definition of the function using some arbitrary types, (as place holders)
 - a **parameterized** definition
- can be considered the definition for a set of overloaded versions of a function
- is identified by the keyword **template**
 - followed by comma-separated list of **parameter** identifiers (each preceded by keyword **class** or keyword **typename**)
 - enclosed between < and > delimiters
 - followed by the signature the function
- Note that every template parameter is a **built-in type** or **class – type** parameters

NPTEL MOOCs Programming in C++ Partha Pratim Das 9

So, this kind of you know pitfalls make macros quite unusable and we lead to what is known as templates. Templates are nothing but functions where the type is parameterized. So, it is a parameterized definition. So, functions have multiple parameters, arguments, formal arguments, and a written type, so all of these or sum of these could be made into parameterized types. So, that you do not really say what the type is, but rather you put a type variable in place of that. So, this is a description of what the syntax how the syntax should be written we will skip over, because once you see the example, it will become obvious us to how you write that.

(Refer Slide Time: 10:50)

The slide has a blue header with the title "Max as a Function Template*" and a small logo on the left. The main content area contains C++ code for a function template. The code defines a template class T and a member function Max(T x, T y) that returns the greater of x and y. It then shows two instantiations of this template: one for int (main function) and one for double (dMax). Both print statements output the value 3.7. A note at the bottom states: "★ Max. now. knows the type" and "★ Template type parameter T explicitly specified in instantiation of Max<int>, Max<double>". The footer includes the NPTEL logo, the course name "NPTEL MOOCs Programming in C++", the professor's name "Partha Pratim Das", and the slide number "10".

```
#include <iostream>
using namespace std;

template<class T>
T Max(T x, T y) {
    return x > y ? x : y;
}

int main() {
    int a = 3, b = 6, iMax;
    double c = 2.1, d = 3.7, dMax;

    iMax = Max<int>(a, b);
    cout << "Max(" << a << ", " << b << ") = " << iMax << endl;
    // Output: Max(3, 6) = 6

    dMax = Max<double>(c, d);
    cout << "Max(" << c << ", " << d << ") = " << dMax << endl;
    // Output: Max(2.1, 3.7) = 3.7

    return 0;
}

★ Max. now. knows the type
★ Template type parameter T explicitly specified in instantiation of Max<int>, Max<double>
```

So, why you want to write the max function against, the max if I write it with int then it looks like int x, int y it will written an int type and the code certainly is returned x greater than y question mark x :: y. So, this is my code of max all that we are doing now is we are saying that we write this without any specific type int, but rather we will use a placeholder in place of int, which is type T and so that is what we write here. And the fact that we are using a placeholder is specified by this template is a keyword and within the corner bracket you write class T this, defines that T is a type variable T is a placeholder for a type and then I can use it.

So, what it does is something interesting, once I have written that then I can use the max function like this I can, I will write max within corner bracket I will put a type and then I will show the original name of the function is max that is templatize name and this is the instantiated name, which means that if I when I put this int it means that used the code of max, but take T to be int. And generate this whole code of the function taking T to be int and then you call on to that function. So, when I later on say max double. So, what it does it takes again generates another code were t is taken to be double a similar code of max is generated and that is compiled .

So, basically once we instantiated int and double actually what I get, when I execute are

two overloaded functions both of which are called max both of which takes two parameters and written one value and all of these for a for a one particular function is have the same types. But one is for two integers written integers one is for two doubles and written the double, so that is, what is the basic job of the templates. So, here a earlier we were having to write this overloads earlier we were having to a macro for these overloads, but now we can written one template function a with the parameterized type and actually get that required functions generated whenever we need them.

(Refer Slide Time: 13:26)

**Max as a Function Template;
Pitfall "Side Effects" – Solved**

```
#include <iostream>
using namespace std;

template<class T>
T Max(T x, T y) {
    return x > y ? x : y;
}

int main() {
    int a = 3, b = 5, iMax;
    // Side Effects
    cout << "Max(" << a << ", " << b << ")" << endl;
    iMax = Max(int)(a+1, b+1);
    cout << iMax << endl;
    // Output: Max(3, 5) = 5
    cout << "a = " << a << ", b = " << b << endl;
    // Output: a = 4, b = 6
}

```

★ Max is now a proper function call – no side effect!

NPTEL MOOCs Programming in C++ Partha Pratim Das 11

So, if we do this then a certainly the problems get solved, we can now go and write the same code, you will find that you are getting the desired effect. Now there is I mean there is nothing special in this, because this is become truly a function. So, this will actually take the value of a take the value of b, call the function and after that increment a, and b individually and that will certainly give the correct results. So, are this does not have the issue that macros was showing.

(Refer Slide Time: 13:50)

The slide title is "Max as a Function Template: Pitfall "C-String Comparison" – Solved". It features a sidebar with a logo and navigation links for Module 3B, Partha Pratim Das, Objectives & Outcome, What is a Template?, Function Template Definition, Instantiation, Specialization, Derivation, Examples, Hypothesis, Summary, and Home. A photo of Partha Pratim Das is on the left.

```
#include <iostream>
#include <cstring>
using namespace std;

template<class T>
T Max(T x, T y) { return x > y ? x : y; }

template<> // Template specialization for 'char*' type
char *Max<char *>(char *x, char *y) { return strcmp(x, y) > 0 ? x : y; }

int main() {
    char *a1 = new char[8], *a2 = new char[8];

    strcpy(a1, "black"); strcpy(a2, "white");
    cout << "Max(" << a1 << ", " << a2 << ") = " << Max<char*>(a1, a2) << endl;
    // Output: Max(black, white) = white

    strcpy(a1, "white"); strcpy(a2, "black");
    cout << "Max(" << a1 << ", " << a2 << ") = " << Max<char*>(a1, a2) << endl;
    // Output: Max(black, white) = white
}

return 0;
}
```

Notes at the bottom:

- Generic template code does not work for C-Strings as it compares pointers, not the strings pointed by them
- We provide a specialization to compare pointers using comparison of strings
- Need to specify type explicitly in instantiation

NPTEL MOOCs Programming in C++ Partha Pratim Das 12

Similarly if we want to work with, also use this templates to compare to c strings and we saw that comparing c strings are difficult, because when we want to compare just if we compare the two pointers then you will not get the same results. So, if we just take this as char star take T as char star I say max char star, then whatever I get will not be a correct thing, because it will give me the same issue as the macros was showing that we will just compare the pointers. So, what can I do is something very, very interesting is I can do what is known as templates specialization, that is I can say that this is a definition of the template max for a type variable T, for any T what so ever. But if the T is specifically char star then I have a different definition of this function.

So, you will look at this, this is a general template the generic or the primary templates as it is called and this is specialized templates you here are actually replacing T by char star, we are replacing T by char star. And since we have replaced c this is become the written type this is become the name of the function because showing that you have replaced T by char star this is T replaced by char star and so on. And since the template had only one template parameter and that has been replaced. So, now, the template specialization list as no parameter at all showing that it is being specialized here. So, what it means that, now if will invoke instantiate max int, it will invoke this version of the template function with T being int. But if we invoke a as we do here max this should be char star actually it

not char char star, if we do max char star there is small typo which I will correct later on.

So, that will actually invoke the specialized template version of this function it will not invoke this one. So, in this version now we take care of the fact that we are not interested to compare the pointers where interested to compare the function that are pointed to by the pointers. So, we do strcmp on these two pointers compare them and give the results. So, it will take care of very easily it will take care of the issue that macros are showing, which was not possible to do in terms of the macro we could not we can gave only one definition of a macro not a multiple once. But here we can actually provide a specialized definition for the template to take care of such issue.

(Refer Slide Time: 16:46)

Max as a Function Template:
Implicit Instantiation

```
#include <iostream>
using namespace std;

template<class T>
T Max(T x, T y) {
    return x > y ? x : y;
}

int main() {
    int a = 3, b = 5, iMax;
    double c = 2.1, d = 3.7, dMax;

    iMax = Max(a, b); // Type 'int' inferred from 'a' and 'b' parameters types
    cout << "Max(" << a << ", " << b << ") = " << iMax << endl;
    // Output: Max(3, 5) = 5

    dMax = Max(c, d); // Type 'double' inferred from 'c' and 'd' parameters types
    cout << "Max(" << c << ", " << d << ") = " << dMax << endl;
    // Output: Max(2.1, 3.7) = 3.7

    return 0;
}
```

■ Often template type parameter **T** may be inferred from the type of parameters in the instance
■ If the compiler cannot infer or infer wrongly, use explicit instantiation

So templates are a good way to solve this. But certainly what will bother you is now the way you invoke the templates is changed. Now you have to invoke them as and the instantiation as we say as to say what the type is or for doing it as a double you have to say what the type is now as it. So, happens that C++ does allow you to skip this in many cases. You may not require to specify the type in the instantiation provided the types of the parameters can tell you what this template variable type must be. So, the idea is like this, that let us say if you are looking into here, the type is not specified here looking at max(a, b). So, what can you get from max ab, if max ab has to match this then you know

that the first hypothesis is this is has got a type int, this has got a type int. So, these are first parameters. So, this is the placeholder for x.

So, if this has to match then T should be int these a second parameter if this has to match then T as to be int, and they are consistent, so if I just put T as int and generate a version of the function which T as int that will match up with this instance. Whereas, when I do max (c,d) then c has double, so is d. So, here making T double will actually make it match the definition of the template. So, earlier we needed to write max int a b which is called the explicit instantiation. Here, we were writing max double cd which is the explicit instantiation we were saying that what types we want, we can skip this now because that can be inferred from the parameter types.

Now, it is possible that in some cases you would not be able to do that. For example, a in this context, I can say that if you have max a,c suppose you want to call max ac and your expectation is this is int, this is double. And certainly int can be implicitly converted to double, but in this case the compiler will not be able to deduce the template type from this. Because from here it gets T as int; from here it gets T as double, so this is contradictory it has to get the same type for both.

But you will able to still call this, for example, if you say max double a c, how will that work because if you say max double if you are explicit in the instantiation then T is double, which means that this is double as well as this is double. Whereas actually this is int this is double the second one is double it directly matches this is int, an int can always be cast implicitly to double as you know. So, it will cast implicitly and do that. So, it says that it is possible that not in every case you will be able to do and implicit instantiation, but in many cases, you would be able to do that. In cases, you cannot because of you want to mean something different or you want to mix up certain types and do some conversion, you will have to use the explicit instantiation.

(Refer Slide Time: 20:20)

The slide title is "Template Argument Deduction: Implicit Instantiation". The sidebar on the left lists navigation links: Module 3B, Partha Pratim Das, Objectives & Outline, What is a Template?, Function Templates, Deduction Instantiation, Template Argument Deduction Examples, Hypotheses, and Summary. A video player in the center shows a man speaking. The main content area contains bullet points and code examples:

- Each item in the template parameter list is a template argument
- When a template function is invoked, the values of the template arguments are determined by seeing the types of the function arguments

```
template<class T> T Max(T x, T y);
template<char*> char *Max<char*>(char *x, char *y);
template <class T, int size> Type Max(T x[size]);
```

```
int a, b; Max(a, b); // Binds to Max<int>(int, int);
double c, d; Max(c, d); // Binds to Max<double>(double, double);
char *a1, *a2; Max(a1, a2); // Binds to Max<char*>(char*, char*);
```

```
int parr[0]; Max(parr); //Error!
```

- Three kinds of conversions are allowed
 - L-value transformation (for example, Array-to-pointer conversion)
 - Qualification conversion
 - Conversion to a base class instantiation from a class template
- If the same template parameter are found for more than one function argument, template argument deduction from each function argument must be the same

NPTEL MOOCs Programming in C++ Partha Pratim Das 14

So, here I have shown a couple of examples of things that you can do implicitly of the different max forms that you had seen. And you will have to be, you will be able to figure out has to how they are deducing the type. Normally, when you use templates and three kinds of conversions are allowed for this types; one is called L-value transformation that is in place of an array, you could present the pointer and vice versa.

You can do a qualification conversion which is basically conversion using a non-const value in place of a const value and so on. And you can do a base class instantiation from a class template which is basically doing an up-cast conversion in the template. So, these are the conversions are allowed; otherwise, the template arguments will have to otherwise follow the strict matching of types and it will not deduce anything. It will say that I am confused I am failing and then you will have to explicitly specify the template conversion argument conversion.

(Refer Slide Time: 21:28)

The slide title is "Max as a Function Template With User-Defined Class". The sidebar on the left lists the following objectives:

- Module 3B
- Partha Pratim Das
- Objectives & Outcome
- What is a Template?
- Function Template
- Default Instantiation
- Template Argument Deduction Example
- Hypothetical Examples
- Summary

The main content area contains the following C++ code:

```
#include <iostream>
#include <cmath>
#include <cstring>
using namespace std;

class Complex { double re_; double im_; public:
    Complex(double re = 0.0, double im = 0.0) : re_(re), im_(im) {}
    double norm() const { return sqrt(re_*re_ + im_*im_); }
    friend bool operator<(const Complex c1, const Complex c2) {
        return c1.norm() > c2.norm();
    }
    friend ostream operator<<(ostream& os, const Complex& c) {
        os << "(" << c.re_ << ", " << c.im_ << ")";
        return os;
    }
};

template<class T> T Max(T x, T y) { return x > y ? x : y; }

int main() { Complex c1(2.1, 3.2), c2(6.2, 7.2);
    cout << "Max(" << c1 << ", " << c2 << ") = " << Max(c1, c2) << endl;
    // Output: Max((2.1, 3.2), (6.2, 7.2)) = (6.2, 7.2)
    return 0;
}
```

Below the code, there are three bullet points:

- When Max is instantiated with class Complex, we need comparison operator for Complex.
- The code, therefore, will not compile without bool operator<(const Complex&, const Complex&).
- Traits of type variable T include bool operator<(T, T) which the instantiating type must fulfill.

At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" with page number "15".

Now, continuing with our max example, I show here that you can use the similar template for user-defined types also complex is a user-defined type here. And I have variables of complex type and I am using max on them with implicit instantiation. Now, of course, you will have to keep in mind that if you do this, then certainly which version; this version is specifically for char star. So, if you are doing calling this then it is inferring c1 is complex; c2 is complex, so it is inferring T is complex for this instantiation. So, it is basically looking at this version. If it is looking at this version, if it is looking at this version, that means, that it is looking at complex, so this is how the generated function will look like complex x, complex y return then x greater than y question mark x y right. This is what it will return.

Now, if it is doing that then what it will require what is x greater than y. It is x is complex, y is complex, so x greater than y will have to be defined for the complex type, if the complex type does not have a x greater than y, that is if the greater than operator has not been overloaded in the complex class. You will not be able to put this instantiation. So, for that, I had to provide a overloading opera of operator greater than in the complex class, if you remove this you will find that this code will not compiled because the it will deduce T to be complex and then it will say that it requires a greater than operator which does not exist in the complex class. And therefore, this templated

version cannot be instantiated for complex.

So, you will have to give the parameter type T there are certain basic operations and operations that T will have to satisfy. So, that if you want to pass some user-defined type want to instantiate with some user-defined type your user-defined type will have to satisfy those properties. For example, the other one is certainly it returns by value. So, this will have to your complex class need to have a copy constructor. Now, here I have not provided one. So, it is assumed that since this does not have dynamically allocated memory it will get a free constructor; free copy constructor from the compiler and that that itself will be used. So, in this way you could use any of the user-defined types also provided they satisfy this trait of T that is the required operations of the type variable T, if the satisfy that then you will be able to use that in terms of instantiating your template.

(Refer Slide Time: 24:32)

The slide title is "Max as a Function Template Overloads". The content is as follows:

```
#include <iostream>
#include <string>
using namespace std;

template<class T> T Max(T x, T y) { return x > y ? x : y; }

template<char*> char *Max<char *>(char *x, char *y) { return strcmp(x, y) > 0 ? x : y; }

template<class T, int size> T Max(T x[size]) { T t = x[0];
    for (int i = 0; i < size; ++i) { if (x[i] > t) t = x[i]; }
    return t;
}

int main() {
    int arr[] = { 2, 6, 3, 7, 9, 4 };
    cout << "Max(arr) = " << Max<int, 7>(arr) << endl; // Output: Max(arr) = 9
    return 0;
}
```

Notes:

- Template function can be overloaded
- A template parameter can be non-type (int) constant

And finally, you can explicitly overload a template function also. So, as we have explained that the template itself is an arbitrary number of overloaded functions because for every type T this is overloaded function of T. Similarly, this is an explicit specialization, this is a specialization for char star, there could be specialization for other types also these are also overloads. But I could have a more classical overload like this for example, here where I have a max which takes an array as a single parameter and that

array is specified by a size. So, you look at this is an array and I am trying to call this max this certainly does not match with this template function, it does not match with this template function both of them need two parameters, but this has a single parameter. So, these are overload of the max template function, and this itself is another template.

And what is interesting about it is; a class type and this also as another template parameter which is called non-type parameter, this is not a here you accept a type and here you get a more traditional function kind of time, this is called a non-type parameter. So, this is possible only if this is an int type. So, you can pass a constant value here, in terms of the template you can pass a constant value here. So, here I am using this non-type parameter to pass the size of the array. So, we will say that the size of the array is 7. So, we are specifying that, so the array if you see this is 7. So, I am passing this here. So, this will tell the template function that the size is 7, so that size is being used within the code of the template function. So, in this way, you can use non-type parameters and as well as explicitly overload the template function with other templates or other non template functions as well.

(Refer Slide Time: 26:43)

Swap as a Function Template

```
#include <iostream>
#include <string>
using namespace std;

template<class T> void Swap(T& one, T& other)
{
    T temp;
    temp = one; one = other; other = temp;
}

int main()
{
    int i = 10, j = 20;
    cout << "i = " << i << ", j = " << j << endl;
    Swap(i, j);
    cout << "i = " << i << ", j = " << j << endl;

    string s1("abc"), s2("def");
    cout << "s1 = " << s1 << ", s2 = " << s2 << endl;
    Swap(s1, s2);
    cout << "s1 = " << s1 << ", s2 = " << s2 << endl;
}

return 0;
}
```

- * The traits of type variable T include
 - default constructor (T():T())
 - copy assignment operator (T operator=(const T&))
- * Our template function cannot be called swap, as std::namespace has such a function

NPTEL MOOCs Programming in C++ Partha Pratim Das 17

So, this is just for your practice and understanding. I will not go through these; it is here where we have written the swap code. Swap is something that you need to do every time

and we have seen that. With the use of reference parameter swap really becomes efficient to write in C++. And now you can make it more efficient by actually writing a swap for any class what so ever except the fact that certainly swapping need copying values. So, if we have to swap then the copy operators are copy constructor in the copy assignment operator has to be there in the available trait of the function parameter; a type parameter T.

(Refer Slide Time: 27:23)

The screenshot shows a presentation slide titled "typename Keyword". The slide content is as follows:

- Module 38
- Partha Pratim Das
- Objectives & Outline
- What is a Template?
- Function Templates
- Template Instantiation
- Template Argument Substitution
- Example
- typename**
- Summary

● Consider:

```
template <class T> f (T x) {  
    T::name * p;  
}
```

● What does it mean?

- T::name is a type and p is a pointer to that type
- T::name and p are variables and this is a multiplication

● To resolve, we use keyword typename:

```
template <class T> f (T x) { T::name * p; } // Multiplication  
template <class T> f (T x) { typename T::name * p; } // Type
```

● The keywords class and typename have almost the same meaning in a template parameter

● typename is also used to tell the compiler that an expression is a type expression

NPTEL MOOCs Programming in C++ Partha Pratim Das 18

Ah finally, let me introduce another keyword typename, like we are writing class here in state C++ also allows us to write the keyword typename. You can interchangeably write class and typename, there is basically no difference except for one case which is illustrated here suppose you have written this. Now, the question is what is this code mean? You can read it in two ways one is you can think of that, this whole thing is name of a variable. So, T is some class name or some you know namespace name. So, the T colon colon name basically turns out to be the name of a variable and this is another variable. So, this is a multiplication, or it could be this is name is basically within T, class T name is a typedef. So, it is a type. So, this is if this is a type then this p is a declaration or pointer declaration of type T colon colon name.

So, from the syntax you cannot figure out which one your meaning that is what the C++

gets confusing. So, C++ has to provide this specific keyword typename to resolve for example, if you just write this then the default interpretation is this is a multiplication. So, this is a variable, this is another variable. But if you want to say that this is a type, and this is a declaration then you have to write typename in front of it. And then you will have to write this where in it will be understood that this is a typename and this is a name of a variable declaration, a pointer variable declaration in this case.

So, whenever you need to use specifically say in the template that you are talking about a type you use this typename keyword, you cannot use a class here to mean the something, because that will get confused with the nested class because if you have a class within a scope then it means another different class definition. So, the new keyword was required, but otherwise in every other context class and typename can be used interchangeably.

(Refer Slide Time: 29:21)

The slide is titled "Module Summary" and features a blue header bar with the NPTEL logo and navigation icons. The main content area has a white background. On the left, there is a sidebar with a vertical menu:

- Module 38
- Partha Pratim Das
- Objectives & Outline
- What is a Template?
- Function Template
- Definition Instantiation
- Template Arguments
- Conclusion
- Example
- Exercises
- Summary

Below the sidebar is a circular profile picture of Partha Pratim Das. The main content area contains the following text and bullet points:

Module Summary

- Introduced the templates in C++
- Discussed function templates as generic algorithmic solution for code reuse
- Explained templates argument deduction for implicit instantiation
- Illustrated with examples

NPTEL MOOCs Programming in C++ Partha Pratim Das 19

So, with this, we are at the end of today's module. We have introduced templates in C++ and specifically we have discussed about function templates and shown how they are advantages over simple overloads and use of macros in C. And how do they allow you to write a generic code meta programming code which given the instantiation which could be implicit or explicit can generate the actual typed function code and that can be used subsequently in your program.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 55
Template (Function Template): Part II

Welcome to module 39 of Programming in C++. We have been discussing about Templates or Generic Programming, Meta Programming in C++ where we could write some functions, templated functions, parameterized functions with one or more type variables so that based on the use either explicitly or implicitly different functions of different parameterized types can get generated as overloads and can get invoked. We have seen a depth in the last module. We have seen at depth an example of a max function which we first wrote in the templated form used it for int and double and then we specialized it for C strings and then we showed that it will work also for user defined types like complex.

(Refer Slide Time: 01:35)

The screenshot shows a presentation slide titled "Module Outline". The slide has a blue header bar with the title. Below the header, there is a sidebar on the left containing a navigation menu with various links. The main content area contains a bulleted list of topics under the heading "Function Template".

- What is a Template?
- Function Template
 - Function Template Definition
 - Instantiation
 - Template Argument Deduction
 - Example
- typename
- Class Template
 - Class Template Definition
 - Instantiation
 - Partial Template Instantiation & Default Template Parameters
 - Inheritance

At the bottom of the slide, there is footer text: "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

In the current module, we will continue on that and our focus would be the other kinds of templates that C++ has, which is known as the Class Template. This is the outline and the blue part is what we discuss will be available on the left of your screen.

(Refer Slide Time: 01:48)

The slide has a blue header with the title 'What is a Template?: RECAP (Module 38)'. On the left, there's a sidebar with navigation links: 'Module 39', 'Partha Pratim Das', 'Objectives & Outline', 'What is a Template?', 'Function Template', 'Class Template', 'Definition', 'Implementation', 'Partial Template', 'Default Template', 'Placeholder', and 'FAQ'. A circular profile picture of Partha Pratim Das is also present. The main content area contains bullet points about templates:

- Templates are specifications of a collection of functions or classes which are parameterized by types
- Examples:
 - Function search, min etc.
 - The basic algorithms in these functions are the same independent of types
 - Yet, we need to write different versions of these functions for strong type checking in C++
 - Classes list, queue etc.
 - The data members and the methods are almost the same for list of numbers, list of objects
 - Yet, we need to define different classes

This is just for a quick recap, this is what we have seen what is a template and we have seen the function part of it.

(Refer Slide Time: 02:00)

The slide has a blue header with the title 'Function Template: Code reuse in Algorithms: RECAP (Module 38)'. The sidebar is identical to the previous slide. The main content area contains bullet points about function templates:

- We need to compute the maximum of two values that can be of:
 - int
 - double
 - char * (C-String)
 - Complex (user-defined class for complex numbers)
 - ...
- We can do this with overloaded Max functions:

```
int Max(int x, int y);
double Max(double x, double y);
char *Max(char *x, char *y);
Complex Max(Complex x, Complex y);
```

With every new type, we need to add an overloaded function in the library!
- Issues in Max function
 - Same algorithm (compare two values using the appropriate operator of the type and return the larger value)
 - Different code versions of these functions for strong type checking in C++

And in terms of the function template we have seen that function templates basically are code reuse in Algorithms. So, you have search algorithm, we have sort algorithm, we

have min algorithm, we have average algorithm and so on. In C++ the code for this is to be return based specifically on the element type. But, in general the algorithm does not change based on the element types so using template we can write function templates which can write this function codes of sorting, searching, min, max, average, all those in a generic form and then instantiate based on that type.

(Refer Slide Time: 02:44)

The slide content is as follows:

- Solution of several problems needs stack (LIFO)
 - Reverse string (char)
 - Convert infix expression to postfix (char)
 - Evaluate postfix expression (int / double / Complex ...)
 - Depth-first traversal (Node *)
 - ...
- Solution of several problems needs queue (FIFO)
 - Task Scheduling (Task *)
 - Process Scheduling (Process *)
 - ...
- Solution of several problems needs list (ordered)
 - Implementing stack, queue (int / char / ...)
 - Implementing object collections (UDT)
 - ...
- Solution of several problems needs ...
- Issues in Data Structure
 - Data Structures are generic - same interface, same algorithms
 - C++ implementations are different due to element type

NPTEL MOOCs Programming in C++ Partha Pratim Das 6

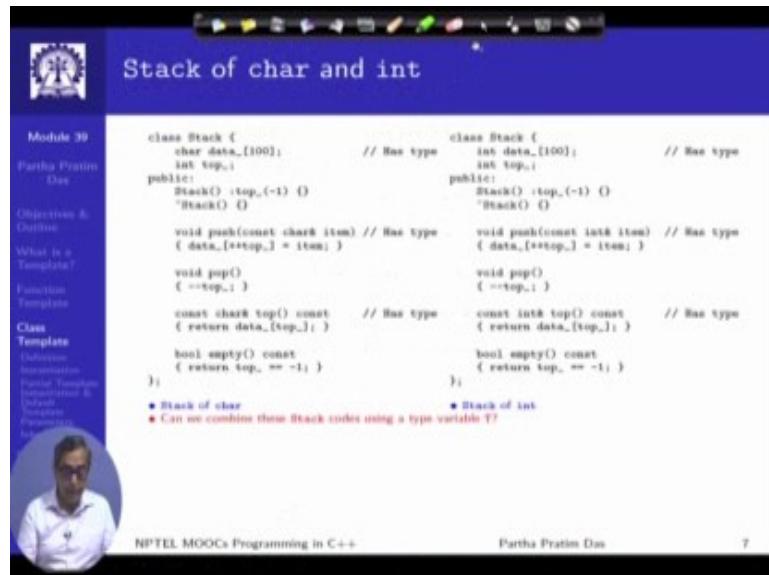
Now, we can do more if we look into code reuse in terms of data structure. For example, consider a stack, the last in first out. There are several problems which you will stack for example, reversing extreme need a stack of character. Converting and infix expression to postfix requires again a stack of characters. Evaluation of postfix expressions might require integer, double, complex different kinds of types that we want to evaluate. The depth first traversal of a tree would need a stack of node pointer types of the three nodes. There could be several problems which need stacks of various different types to be used for a specific problem solution.

Now, one choice is to write a stack class for each one of this type whenever we need, but what we are looking at can we generically have a stack code which can be instantiated given the particular type that we want, because, the stack as a concept is a last in first out with a set of few interfaces like, push, pop, top, empty and so on, which does not change

depending on the specific element type that the stack is using. And if you look further you will find similar commonality with queue use the task scheduling process scheduling requiring queue user several problems that need list like implementing stack queue all those then any kind of object collections and so on and so forth.

The class templates are a solution to such code reuse where, we identify the generic part of a data structure where you have the same interface and same or very close algorithms, similar algorithms but the implementations need to be different due to the element types can we combine them in terms of a common generic class template.

(Refer Slide Time: 04:50)



Stack of char and int

```

Module 39
Partha Pratim Das

Objectives & Outline
What is a Template?
Function Templates
Class Templates
    Definition
    Instantiation
    Partial Specialization
    Default Arguments
    Templates
    Overloading
    Inheritance
    etc.

Partha Pratim Das

```

```

class Stack {
    char data_[100];           // Base type
    int top_;                  // Base type
public:
    Stack() : top_(-1) {      // Base constructor
        *this = {};
    }
    void push(const char item); // Base type
    { data_[++top_] = item; }

    void pop();
    { --top_; }

    const char top() const;   // Base type
    { return data_[top_]; }

    bool empty() const;
    { return top_ == -1; }
};

• Stack of char
• Can we combine these stack codes using a type variable T?

```

```

class Stack {
    int data_[100];           // Base type
    int top_;                  // Base type
public:
    Stack() : top_(-1) {      // Base constructor
        *this = {};
    }
    void push(const int item); // Base type
    { data_[++top_] = item; }

    void pop();
    { --top_; }

    const int top() const;   // Base type
    { return data_[top_]; }

    bool empty() const;
    { return top_ == -1; }
};

• Stack of int

```

NPTEL MOOCs Programming in C++ Partha Pratim Das 7

So, just to illustrate this is a left and right, if you just look in here this is a stack of character, which is character. These are just shown as comments the particular code lines which need the knowledge of the type and this is a stack of integer, so these are the lines which need. So you have integer here, char here, you have char here, you have char here, and int here.

(Refer Slide Time: 05:25)

The slide has a blue header with the title 'Class Template'. On the left, there's a sidebar with navigation links: Module 39, Partha Pratim Das, Objectives & Outline, What is a Template?, Function Template, Class Template Definition, Instantiation, Partial Template Definition & Default Parameters, and Acknowledgements. A small profile picture of Partha Pratim Das is also on the sidebar.

The main content area contains a bullet-point list under the heading '● A class template':

- describes how a class should be built
- Supplies the class description and the definition of the member functions using some arbitrary type name, (as a place holder)
- is a:
 - parameterized type with
 - parameterized member functions
- can be considered the definition for a **unbounded set** of class types
- is identified by the keyword **template**
 - followed by comma-separated list of parameter identifiers (each preceded by keyword **class** or keyword **typename**)
 - enclosed between **<** and **>** delimiters
 - followed by the definition of the class
- is often used for **container classes**
- Note that every template parameter is a **built-in type** or **class – type parameters**

At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

Other than that the rest of the code is exactly the same so why not we replace this by a type variable like we did in case of the function. This is what leads to the class template which is parameterized with type and may have parameterized member functions. Rest of the definition is for the details and will look at the example.

(Refer Slide Time: 05:46)

The slide has a blue header with the title 'Stack as a Class Template'. On the left, there's a sidebar with the same navigation links as the previous slide. A small profile picture of Partha Pratim Das is also on the sidebar.

The main content area shows the code for 'Stack.h':

```
template<class T>
class Stack {
    T data_[100];
    int top_;
public:
    Stack() : top_(-1) { }
    ~Stack() { }

    void push(const T& item)
    { data_[++top_] = item; }

    void pop()
    { --top_; }

    const T& top() const
    { return data_[top_]; }

    bool empty() const
    { return top_ == -1; }
};
```

Below the code, there are three red bullet points:

- Stack of type variable T
- The traits of type variable T include copy assignment operator (`T operator=(const T&)`)
- We do not call our template class as stack because std namespace has a class stack

At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

So for the stack all that we do we parameterized this type element type as T. As you do that as you we can see the places where you need is when you push I need to know the element type which is T when I do a top I need to know the element type, pop does not need to know it, empty does not need to know that. Since this type is T, I parameterized and exactly the way I had done in case of function I put a template here in terms of template class T saying that this is a template variable here. And that template variable is used in terms of these member functions.

So, this is what makes it a stack which is templatized, which can be instantiated for anything. Of course, for this stack template to work we will need the type T, the type variable T to satisfy certain property certain traits. For example, item is of type T and data i is of type T and data element. So we see that there is an assignment possible here. The copy assignment operator must be possible in this place without that you will not be able to instantiate the stack with a given particular type.

(Refer Slide Time: 07:14)

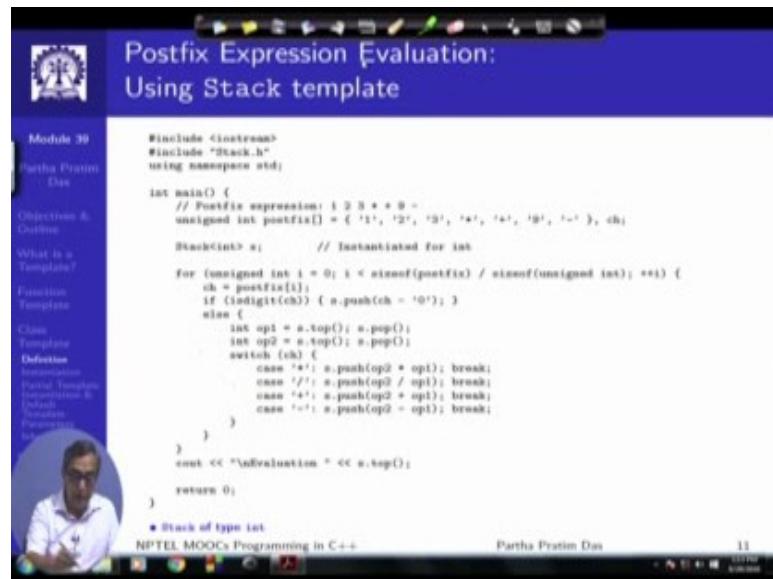
```
#include <iostream>
#include "Stack.h"
using namespace std;

int main() {
    char str[10] = "ABCDE";
    Stack<char> s;           // Instantiated for char
    for (unsigned int i = 0; i < strlen(str); ++i)
        s.push(str[i]);
    cout << "Reversed String: ";
    while (!s.empty()) {
        cout << s.top();
        s.pop();
    }
    return 0;
}
* Stack of type char
```

If we look at this now using this, assuming that all that goes into the stack dot h the header, then I just instantiate it much like the way we are instantiating the function we just instantiate it say it for the character. This will now give me a stack of characters

which I can use this. I will not go into explaining this code we have seen this number of times in this codes, we can use that stack to actually reverse the strain.

(Refer Slide Time: 07:48)



Postfix Expression Evaluation:
Using Stack template

Module 39
Partha Pratim Das

Objectives & Outline
What is a Template?
Function Template
Class Template Definition
Instantiation
Partial Template Instantiation B
Default Template
Specialization

```
#include <iostream>
#include "Stack.h"
using namespace std;

int main() {
    // Postfix expression: 3 2 3 * + 9 =
    unsigned int postfix[] = { '3', '2', '3', '*', '+', '9' };
    char ch;

    Stack<int> s; // Instantiated for int

    for (unsigned int i = 0; i < sizeof(postfix) / sizeof(unsigned int); ++i) {
        ch = postfix[i];
        if (isdigit(ch)) C s.push(ch - '0');
        else {
            int op1 = s.top(); s.pop();
            int op2 = s.top(); s.pop();
            switch (ch) {
                case '*': s.push(op2 * op1); break;
                case '/': s.push(op2 / op1); break;
                case '+': s.push(op2 + op1); break;
                case '-': s.push(op2 - op1); break;
            }
        }
    }
    cout << "\nEvaluation = " << s.top();
    return 0;
}

* Stack of type int
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 13

With that same header stack dot h. I can now write a separate application. So, this same header that is a same templated stack code I can write a different application to evaluate postfix expression. Since expressions here are of integers so I need a stack which will keep the expression values which of integer, so this is instantiated with int. If I had done with C, I would have required to use, two different stack implementations, the char base implementation for the reverse string and the int reverse implementation for this particular postfix evaluation problem, but I have managed with the same templated stack definition and just instantiated with two different types. That is the basic power of the class template and this gives us lot of generalization in terms of the data structures in particular and different utility classes.

(Refer Slide Time: 08:41)

The screenshot shows a presentation slide titled "Template Parameter Traits". The slide has a blue header bar with the title. Below the header, there is a sidebar on the left containing a navigation menu with items like "Module 39", "Partha Pratim Das", "Objectives & Outline", "What is a Template?", "Function Template", "Class Template", and "Definitions". To the right of the sidebar, the main content area starts with a section titled "Parameter Types" which includes a bulleted list of requirements. At the bottom of the slide, there is footer information including "NPTEL MOOCs Programming in C++", the name "Partha Pratim Das", and the number "12".

Now, naturally as I have mentioned that when we do this instantiation. Earlier we saw it for function templates now you are seen it for class template, we will have to make sure that the parameters the type parameters that are used in the template they will satisfy certain properties that is they may be of any type. Maybe other parameterized types also, they may template type themselves, but what is important is they must support the methods that are required for the implementation of the function template or the implementation of the class. So, those are the traits like, they may require to support constructor, they will require to support different operators and we saw instances of that. Those are the basic type traits that both the function template as well as the class template will need to follow.

(Refer Slide Time: 09:34)

The slide title is "Function Template Instantiation: RECAP (Module 38)". It features a sidebar with navigation links: Module 39, Partha Pratim Das, Objectives & Outline, What is a Template?, Function Template, Class Template Definition Instantiation, Partial Template Instantiation, Default Parameters, and Standard Library Functions. A small portrait of Partha Pratim Das is on the right. The main content area contains bullet points and code snippets:

- Each item in the template parameter list is a template argument
- When a template function is invoked, the values of the template arguments are determined by seeing the types of the function arguments

```
template<class T> T Max(T x, T y);  
template<char*> char *Max<char *>(char *x, char *y);  
template <class T, int size> Type Max(T *[size]);  
  
int a, b; Max(a, b); // Binds to Max<int>(int, int);  
double c, d; Max(c, d); // Binds to Max<double>(double, double);  
char *s1, *s2; Max(s1, s2); // Binds to Max<char *>(char *, char *);  
  
int pval(0); Max(pval); // Error!
```

- Three kinds of conversions are allowed
 - L-value transformation (for example, Array-to-pointer conversion)
 - Qualification conversion
 - Conversion to a base class instantiation from a class template
- If the same template parameter are found for more than one function argument, template argument deduction from each function argument must be the same

At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

So, this is what you saw in case of function template, this is just for your recap.

(Refer Slide Time: 09:42)

The slide title is "Class Template Instantiation". It has the same sidebar as the previous slide. The main content area contains bullet points:

- Class Template is instantiated only when it is required:
 - template<class T> class Stack; is a forward declaration
 - Stack<char> s1; is an error
 - Stack<char> *ps; is okay
 - void ReverseString(Stack<char>& s, char *str); is okay
- Class template is instantiated before
 - An object is defined with class template instantiation
 - If a pointer or a reference is dereferenced (for example, a method is invoked)
- A template definition can refer to a class template or its instances but a non-template can only refer to template instances

At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

In terms of class template usually the instantiation needs to be done explicitly and it is important that since it is a class it is quite possible that, I can define the class as a forward declaration without actually providing, so I can actually just write this which is

called an Incomplete stack type. It just says this is the forward declaration to tell the system that there is a class called stack which is parameterized by type T, but it does not say what the methods are and so on. So, with that, if I try to instantiate the object then I will get an error, because certainly if object cannot be instantiated unless I know the constructor, destructor, other operators and members and so on.

But I can still define a pointer to this type; I can define a reference to this type. So I can define a reverse string function which takes this type as a reference when I actually do not know what that. But once I want to implement the body of the reverse string function when I want to use the stack operations naturally I will need to know what that stack definition actually is.

(Refer Slide Time: 11:09)

The screenshot shows a presentation slide with the following details:

- Title:** Class Template Instantiation Example
- Module:** Module 39
- Speaker:** Partha Pratim Das
- Content:**

```
#include <iostream>
using namespace std;

template<class T> class Stack; // Forward declaration
void ReverseString(Stack<char*> s, char *str); // Stack template definition is not needed

template<class T> // Definition
class Stack { T data_[100]; int top_ = -1; public: Stack() : top_(-1) {} ~Stack() {} void push(const T& item) { data_[++top_] = item; } void pop() { --top_; } const T& top() const { return data_[top_]; } bool empty() const { return top_ == -1; } };
int main() {
    char str[10] = "ABCDE";
    Stack<char*> s; // Stack template definition is needed
    ReverseString(s, str);
    cout << "Reversed String: ";
    while (!s.empty()) { cout << s.top(); s.pop(); }
}
void ReverseString(Stack<char*> s, char *str) { // Stack template definition is needed
    for (unsigned int i = 0; i < strlen(str); ++i) s.push(str[i]);
}
```
- Navigation:** The slide has a standard Windows-style navigation bar at the top.

In terms of instantiation this you know lazy instantiation is something which is often very useful. So, I am just showing the same reverse string code in a little bit different manner, earlier this whole stack class was put into stack dot h enters included here, so as if the whole think was happening there itself. But now I am including it here to show if few things, for example, here we have a forward declaration, so with that forward declaration I can have a signature of the reverse string function which will reverse the

string put in here and because as a reference all that it needs to know that it is using a stack which is templated by T and the template is instance is char in this case.

But it does not know what that type is, what it does not know how the type is implemented and it does not care because it is just looking at a reference. If I have this in my main I can actually invoke this function because all that I need to know is a signature of the function the body can come later on, so I have deliberately put the body at a later point of time just you show that the main does not need to know the body.

But certainly, this needs that I pass the instance of a stack as a reference parameter here. So, main needs to instantiate this stack. While you could define the signature of reverse string without actually knowing the definition of the stack you cannot write the main function, because you cannot write this instantiation unless you know the definition of the stack. So the definition of the stack has to precede the instantiation of the stack. Because now if you have an object instance we must be able to construct, it must be able to destruct, it must be able to invoke all different operations.

So this is the kind of, I just wanted to highlight that in keys of a class template instantiation it is not necessary that you will have to always instantiate everything together. If you are instantiating the reference to the class or a pointer to the templated class then you may not need to know the whole definition of the class, you could just manage with the declaration of the class a forward declaration which says that these is the templated class what are the different types and so on.

(Refer Slide Time: 13:47)

The slide title is "Partial Template Instantiation and Default Template Parameters". The slide content shows a C++ code snippet demonstrating template specialization. The code defines a class `Student` with two template parameters `T1` and `T2`. It includes two versions of the constructor and a `Print()` method. Version 1 uses `int` for `T1` and `string` for `T2`. Version 2 uses `char*` for `T1` and `string` for `T2`. The `main()` function creates objects of both types and prints their details.

```
#include <iostream>
#include <cstring>
using namespace std;

template<class T1 = int, class T2 = string> // Version 1 with default parameters
class Student { T1 roll_; T2 name_;}
public: Student(T1 r, T2 n) : roll_(r), name_(n) {}
void Print() const { cout << "Version 1: (" << name_ << ", " << roll_ << endl; }

template<class T1> // Version 2: Partial Template Specialization
class Student(T1 r, char* n) : roll_(r), name_(strcpy(new char[strlen(n) + 1], n)) {}
void Print() const { cout << "Version 2: (" << name_ << ", " << roll_ << endl; }

int main() {
    Student<int, string> s1(2, "Ramesh"); // Version 1: T1 = int, T2 = string
    Student<int> s2(1, "Shampa"); // Version 1: T1 = int, defa T2 = string
    Student<char*> s3("Ragan"); // Version 1: defa T1 = int, defa T2 = string
    Student<string> s4("99", "Lalita"); // Version 1: T1 = string, defa T2 = string
    Student<int, char*> s5(3, "Gouri"); // Version 2: T1 = int, T2 = char*
    s1.Print(); s2.Print(); s3.Print(); s4.Print(); s5.Print();

    return 0;
}

Version 1: Ramesh, 2
Version 1: (Shampa, 1)
Version 1: (Ragan, 7)
Version 1: (Lalita, 99)
Version 2: (Gouri, 3)
```

This is the basic class template. Next we show something, this is just for your understanding of completeness, I will not get much in depth. This is just to show that it is like in terms of function template we saw that if the `max` function had one type parameter `T` and for `char star` we wanted a different behaviour so he specialized that and replace `T` and just put `char star` and put a different one; function definition for that. This is also possible for class and I show that it is actually possible that if I have more than one parameter then I can partially specialize those parameters, so that is what I am trying to illustrate here.

So, there is the template where the `student` class here which is templatized by two types `T1` and `T2`. `T1` is a type of `roll`, `T2` is the type of `name`. so possibility one could be the role could be an integer it could be a string and so on. `Name` could be a string type in C++ or it could be a `char star`, C string type and so on. These are two different types that we have. So, what you do in that is basically, I am there is not much functionality given, you just do a construction and there is a `print` in which you can print these two fields, so just for illustration.

Now, what is interesting is the next one, where we actually partially specialize this? There are two parameters `T1` and `T2` and I have partially specialize this, I still continue to

have a template which has a parameter T1, but T2 have explicitly put as char star and then I have used. In case of T2, I am using char star I have explicitly put that char star. So this becomes the partial instantiation of the template. Earlier this template of student class needed two types to be specified T1 and T2, this needs only one type to be specified which is T1, the other has already been specialized. In a template definition, when you have specialized for all the type parameters then you say that the template is fully specialized otherwise you say this is partially specialized.

If we look at this some class instances with that, suppose we have created an instance like it in string naturally this specifies both of them. So it is trying to talk about this template with T1 being int and T2 be string. You can easily make out from the output, you have done S1 dot; S1 is a object created for this template version and we are doing S1 dot print that choose version one is being used.

In the second one, what we have used? In the second one, we had used something interesting. In second one what you are saying is, look at this carefully, in this we have also specified something as T1 equal to int or T2 equal to string. Recall that in terms of functions we can have default parameters of functions which are values, we can write int x equal to initialized with 5, so that if I do not pass that parameter it will be taken as 5. Similarly, you can have default type parameters. So if I do not specify this then it will be taken as that type. If I am just saying string student int, if I say student int then it means that I am taking this. And I have not specified T2 which is taken by default to be string. So this is by default taken to be string.

I can do this, which will again mean this, where both of taken default parameters values. Default T1 is int, default T2 is int. I can do a student string then I do student string it means that I have done T1 to be string and T2 is a default which is also string. You can just see I have shown the output that you get generated. At the end, if I say that what did we do here? We said that the second parameter is partially specialized to char star. So if I put int char star then it does not mean this template because the second parameter has been specialized partially, so if I say this then it means this template and you can see we are printing for S5 when we do print S5 dot print you can see that the version two is being printed which shows that you are using the partially specialized template version.

This is just to show you that these kinds of things can be done you can have multiple parameters and partially specialize them as you go from one to the other and you can have default parameters also.

(Refer Slide Time: 19:39)

The slide title is "Templates and Inheritance: Example (List.h)". The content shows the following C++ code:

```
#ifndef __LIST_H
#define __LIST_H

#include <vector>
using namespace std;

template<class T>
class List {
public:
    void put(const T& val) { items.push_back(val); }
    int length() { return items.size(); }
    bool find(const T& val) {
        for (unsigned int i = 0; i < items.size(); ++i)
            if (items[i] == val) return true;
        return false;
    }
private:
    vector<T> items;
};

#endif // __LIST_H
* List is basic container class
```

The sidebar on the left lists topics such as Module 39, Partha Pratim Das, Objectives & Outline, What is a Template?, Function Template, Class Template, Definition, Inheritance, Partial Template Specialization & Default Parameter, and Help. There is also a small portrait of Partha Pratim Das.

Finally, before I end I just show you an example of using template with class inheritance so this is just trying to create a bounded set kind of data structure. There is a list, there is nothing specifically interesting about the list is just a list which has a add function called put, has a length function, has a find function to find a particular element, uses vector internally to keep the elements. So that is kind of a supporting data structure.

(Refer Slide Time: 20:13)

```
#ifndef __SET_H
#define __SET_H

#include "List.h"

template<class T>
class Set {
public:
    Set();
    virtual void add(const T& val);
    int length();
    bool find(const T& val);
private:
    List<T> items;
};

template<class Tp>
void Set<Tp>::add(const T& val)
{
    if (items.find(val)) return;
    items.put(val);
}

template<class T>
int Set<T>::length() { return items.length(); }

template<class T>
bool Set<T>::find(const T& val) { return items.find(val); }

#endif // __SET_H

// Set is a base class for a set
// Set uses List for container
```

Then you define a set using this list. A set has a list of items a class T. It has a virtual function that can add elements find the length. So what will basically do is if you add an element it will go to the list it will go to items and do a put. If you want to this you will do a put here, I think, did I miss anything? no. It will add does this, add actually if you want to add to a set, now this is a set, this is interesting, this is a set so every element has to be unique, set has uniqueness.

The way I do it is, I first take the value on this list I find out if the element belongs to this list if it does belong then this already there in the set so you do not do anything you just return. If I does not belong then the control comes here, then you put it that is add it to the list so this is what. Length is simply a rapper on the list length, find a rapper on the. This gives you a type of having set for any element type.

(Refer Slide Time: 21:40)

```
#ifndef __BOUND_SET_H
#define __BOUND_SET_H

#include "Set.h"

template<class T>
class BoundSet : public Set<T> {
public:
    BoundSet(const T &lower, const T &upper);
    void add(const T &val);
private:
    T min;
    T max;
};

template<class T> BoundSet<T>::BoundSet(const T &lower, const T &upper)
: min(lower), max(upper) {}

template<class T> void BoundSet<T>::add(const T &val) {
    if (find(val)) return;
    if ((val < min) || (val > max))
        Set<T>::add(val);
}

#endif // __BOUND_SET_H

* BoundSet is a specialization of Set
* BoundSet is a set of bounded items
```

Now, suppose I want to have a bounded set. Bounded set by the name here, which is a set having two limits; the elements will have to be within that limit there a bound set will be able to have only members which are within min and max values. So it is a specialization from the set. This you can see how you write the specialization. The bound set is also templatized because it has a template parameter which is an element type, but it specializes from set T. Then the bound set, it has a constructor, it has an add. And, what simply does is when you try to add, it has to change that if is already there, it will return which is the behaviour of the set.

But, if the value is within the min and max then it will go to the parent set class object that is base class object and add that. But if it is not then it does nothing, it will simply ignore you can through an exception here also and do some other behaviour. But what I am trying to just show is here I have a bound set which is a specialization of set, this is templatized, this also is templatized and this has a part of it as a component is has a list which as a part of it has a vector. All of these are templatized, I finally get a bound set which of any type using all these templatized classes and using the inheritance on this templatized class. So, this is how the templates can be mixed up with inheritance feature as well.

(Refer Slide Time: 23:38)

Templates and Inheritance:
Example (Bounded Set Application)

```
#include <iostream>
using namespace std;
#include "BoundedSet.h"

int main() {
    int i;
    BoundedSet<int> bset(3, 21);
    SetInt* setptr = &bset;

    for (i = 0; i < 20; i++) setptr->add(i);

    if (bset.find(4))
        cout << "We found an expected value\n";
    if (bset.find(0) || bset.find(28))
        cout << "We found an unexpected value\n";
    return -1;
} //main
We found an expected value
We found no unexpected value
• Uses BoundedSet to maintain and search elements.
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 20

This is a final bound set application you can just complete the application and check and taken run it that you have added some numbers to this set, and then you try to find four which you are expected to get. And you check if you have of something like 0 or 25 in the list which should not be there because your list is between 3 and 21. So you say there is no unexpected value. This is just shows with the example as to how your bound set data type will work.

(Refer Slide Time: 24:17)

The slide is titled "Module Summary" and is part of "Module 39" by Partha Pratim Das. It features a sidebar with navigation links: "Module 39", "Partha Pratim Das", "Objectives & Outcome", "What is a Template?", "Function Template", "Class Template", "Delegates", "Metaprogramming", "Partial Specialization & Default Arguments", "Programmatic Delegates", and "Delegating Function Objects". The main content area contains a bulleted list of topics covered in the module:

- Introduced the templates in C++
- Discussed class templates as generic solution for data structure reuse
- Explained partial template instantiation and default template parameters
- Demonstrated templates on inheritance hierarchy
- Illustrated with examples

At the bottom left is a portrait of Partha Pratim Das, and at the bottom right are the slide number "28" and the footer "NPTEL MOOCs Programming in C++".

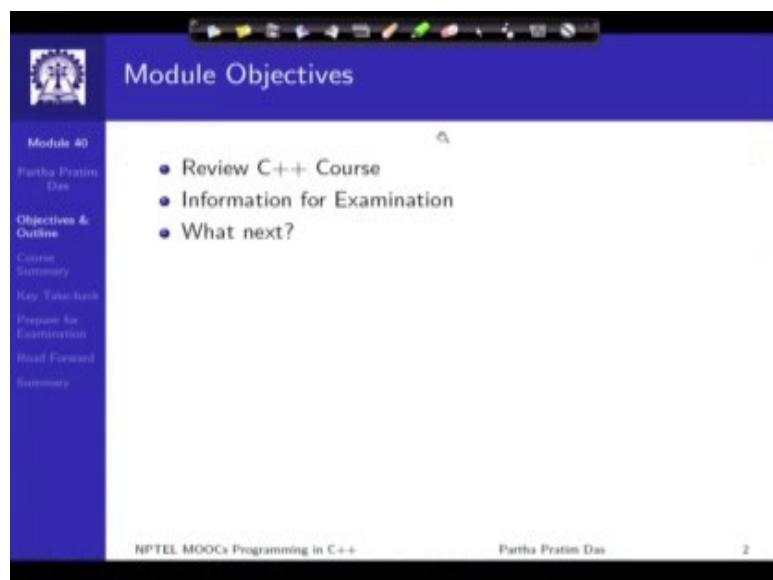
To summarize, we have introduced templates in C++ and we have discussed class template has a generic solution to data structure. Combined with the function template this gives us a major advantage in terms of being able to write generic programming, meta programming code and gives a foundation of what is known as a standard template library or STL of C++.

Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 56
Closing Comments

Welcome to module 40 of Programming in C++. As you know, this is the last module of this current course. So, we are not going to introduce any new material in this module. I would rather summarize the overall course, and try to make some closing remarks for you to move forward.

(Refer slide Time: 00:51)



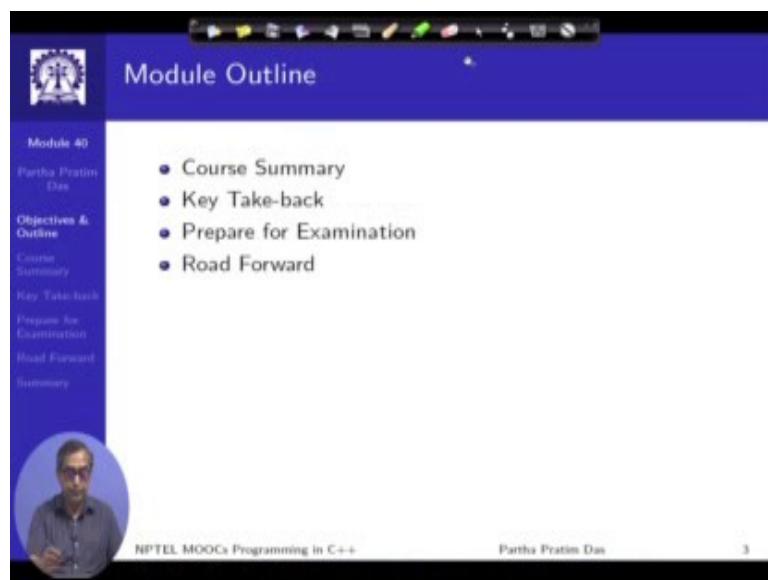
The screenshot shows a presentation slide titled "Module Objectives". The slide has a blue header bar with the title and a white content area. On the left, there is a vertical sidebar with a logo at the top and a list of navigation links: "Module 40", "Partha Pratim Das", "Objectives & Outline", "Course Summary", "Key Take-hack", "Program for Examination", "Road Forward", and "Summary". The main content area contains a bulleted list under the heading "Module Objectives":

- Review C++ Course
- Information for Examination
- What next?

At the bottom of the slide, there is footer text: "NPTEL MOOCs Programming in C++" and "Partha Pratim Das". There is also a small number "2" in the bottom right corner.

So, the objective of this module would be to review quickly what we have done, what we have not done in the C++ course, in these 8 weeks. And I would like to put in few points regarding how we should be preparing for your examination, and what should be your next course of action beyond this course.

(Refer slide Time: 01:21)

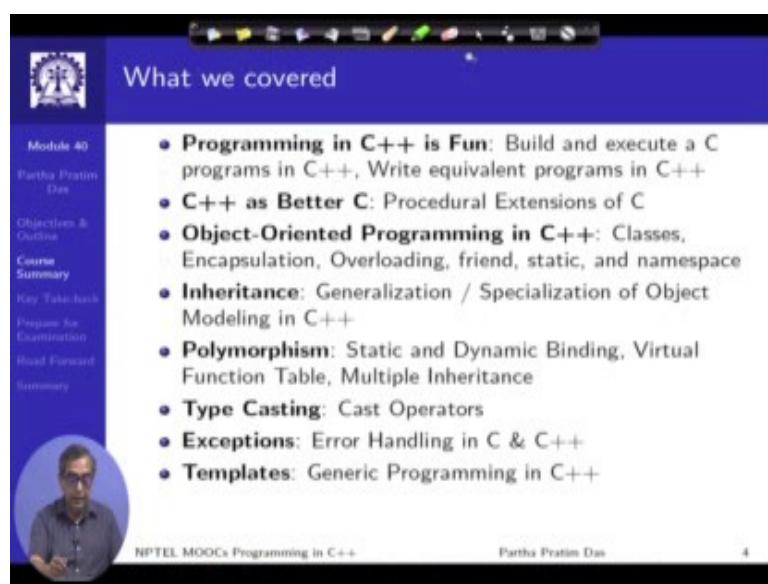


This slide shows the 'Module Outline' for Module 40. The title 'Module Outline' is at the top. On the left, there's a sidebar with a logo, the name 'Partha Pratim Das', and a vertical menu with options: Objectives & Outline, Course Summary, Key Take-back, Prepare for Examination, Road Forward, and Summary. A circular video player window shows a man speaking. At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'. The slide number '3' is in the bottom right corner.

- Course Summary
- Key Take-back
- Prepare for Examination
- Road Forward

So, this is the module outline and we get started here.

(Refer slide Time: 01:26)



This slide shows the 'What we covered' section for Module 40. The title 'What we covered' is at the top. The sidebar is identical to the previous slide. The main content area lists several topics covered in the module:

- **Programming in C++ is Fun:** Build and execute a C programs in C++, Write equivalent programs in C++
- **C++ as Better C:** Procedural Extensions of C
- **Object-Oriented Programming in C++:** Classes, Encapsulation, Overloading, friend, static, and namespace
- **Inheritance:** Generalization / Specialization of Object Modeling in C++
- **Polymorphism:** Static and Dynamic Binding, Virtual Function Table, Multiple Inheritance
- **Type Casting:** Cast Operators
- **Exceptions:** Error Handling in C & C++
- **Templates:** Generic Programming in C++

At the bottom, it says 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'. The slide number '4' is in the bottom right corner.

So, the fun really starts happening when we start coding, the same concepts as in the C program in terms of C++ style. And start using different STLs, and all other different features and writing equivalent program. And through this, we really look into how C++

can improve the overall programming experience. With this foundation, what we covered; what we called C++ as better C that is the procedural features of C are extended in C++ to provide a better procedural programming language. This is required from two perspectives; one is just to make things better to work with, and second the most of these features the procedural extensions are critical to support the object-oriented features.

So, if we can just recall we have taken a look into the const-ness of values and variables which is critical in terms of getting rid of manifest constants being able to maintain better type all through the C++. And also in terms of introducing variety of notions of const-ness, later on in terms of member function and so on. As you know by now is the const-ness turned out to be such an important improvement in the C programming that C language standard actually went forward, and have taken const as a part of C standard back way from C++.

Besides that we saw a whole lot of other features as better procedural features. We talked about reference parameter, we talked about overloading. And very significantly completely new operators for memory management; the operator new and operator delete, the array new, array delete and so on. So, combined with all these, we have a procedural extension of a C, these features do not inherently have any object oriented concept, but these make us C kind of procedural coding in C++ much better.

With that foundation of better procedural support, we moved onto discussing object-oriented programming or OOP in C++. Certainly this was done at a detailed in length; we talked about classes, their members, the access specifiers, constructors, destructors, lifetime and all of these which basically come under the generic encapsulation paradigm of object-oriented programming. Where you encapsulate in two ways, one you encapsulate by packaging multiple data members in terms of an object, and only try to provide a set of behaviour, set of member functions to that. And with a help of visibility, the access specifiers, you make sure as to which part of the object is visible accessible to which members or which agents.

On top of that in support of object-oriented programming, we introduced a whole lot of very important features like overloading of a member functions. We talked about friend

member function, which provide a somewhat a different kind of visibility access than the public or private visibility. We talked about static data members, and member functions. We talked about namespaces. Again namespace in that way may not be consider a strictly object-oriented features, but as it turns out that one when we have to introduce classes they themselves introduce some kind of a namespace. So, a namespace is a logical consequence and certainly it gives a much better source organization code, organization options in terms of C++ programming.

So, with all these together, the main point at up to this part of the course has been that we can with the use of the object oriented features and now empowered to define a data types ourselves. So, the main achievement up to this point in the course where we just to object-oriented encapsulation overloading kind of features is that we can now built our own user-defined data type. And this data types could be as complete in the functionality as the built in data types like int, char, double and so on.

And we saw how all these different features of overloading, and friend functions and operator overloading specifically can be used to really have data types, which can be used to write expressions. We have consistently try to use complex data type as an example. Of course, you can extend it, you can do similar things for other data types as well like fraction like vector matrix and so on.

And built your own required data type, wherever you want and there is no end to what you can achieve in terms of this data types. And certainly while building the data type, there are certain point to keep in mind which I would specifically like to remind you off is a fact that while you bring a data type. And for that, you are overloading the operators; try to keep the semantics of the operator as close to the semantics of the operator in built in types. And do not make it very different, for example, if you are defining a set data type, and you are overloading the plus operator, then it would be intuitively much better to associate the plus operator with union of sets than intersection of sets or say the difference of sets and so on.

At the same time, in terms of the signature, it should remain as close to the built in data type operators as possible in terms of whether it returns the result through value, or

through a reference, or through a constant reference and so on. And we have discussed at length in terms of what difference nuances mean. Other notable points here is a detailed or detailed discussion on copy semantics or detailed discussion on object lifetime and or the variety of free functions that the compiler provide you to support ease of programming.

Moving on next, what you took up you forms another core aspect of object-oriented programming called inheritance or generalization specialization hierarchy. And though this course did not have enough opportunity to discuss object-orientation per say as a design paradigm, but still we have been able to see that in terms of real life world, there is a whole lot of situations where the specialization generalization between the problem domain, within the problem domain, between different concepts is just a natural phenomenon.

And in terms of the inheritance features, in terms of being able to derive a class from a base class, and in that process override the base class member functions with new member function and overload them if required and reuse them if you just inherit and so on. we can provide a whole lot of complex semantics to the natural world in a very organized and aligned manner. And this probably is inheritance probably is one of the key features of C++; particularly of object oriented aspects of C++ coupled with encapsulation, overloading. And inheritance provide you the core frame work in which the object-orientation can be very extensively modelled and programmed and manipulated in terms of the C++ language.

Next, what do we looked at is, what is more commonly called polymorphism. Though the use of the word polymorphism will caution you has a very variety of different meaning in C++, but here what we meant by polymorphism is primarily dynamic binding. This is a very different kind of a feature that we took a look into where the type of a pointer the static type of a pointer or the static type of a reference alone does not decide the particular binding of a member function.

The type of the member or the specific member function being bound with a pointer dereferencing or a reference access is dependent on the actual object being pointed to at

the runtime that is a dynamic type of the pointer, dynamic type of the reference. And that has given us the ability to really build a big class hierarchy with this common base class and with a capability of virtual functions which can do runtime delegation of methods to appropriate runtime objects, the pure virtual function, abstract base class those are very key concepts that were introduced in this part. And we specifically spent time to also understand, how this virtual function mechanism work in terms of the virtual function table and how does it ramifying in terms of the multiple inheritance.

So, these all together form a very strong object-oriented foundation to the C++ language. And then in the next two parts, we spent substantial amount of time in looking into two features; the type casting and exceptions, the features that become critical not only to support better object-oriented programming, but also to overall improve the experience of programming. So, in type casting, we talked about the possible conversion of objects at the runtime, at the static time based on their static type or their dynamic type and so on.

That is what we saw a whole lot of rules, there the implicit casting rules, the C style casting, what are the issues of the C style casting, and why, particularly cast operators the four-cast operators of which the static cast is certainly the most widely used. The most possible kind of static time casting can be done in terms of this static cast or the const cast, which applies to CV qualification or the dynamic cast which applies to the runtime casting particularly for down casting purposes.

So, you can see that the cast operators basically provide whole lot of different semantics based on the different context of a casting. And as you get more and more experience in C++, you will discover for yourself that one particularly now you also know template that actually the cast operators are nothing but template definitions. So, In fact, it is not only that these are the four-cast operators that you have, but it is possible that you can define your own cast operator and define semantics for that.

So, casting or changing the type of an object or basically using an object of one type in the context of another is a very critical features, which is required for supporting strong object orientation, because a strong object orientation has meant see into a very strongly

type language. And therefore, there is often a context of using an object of a certain type in the context of another type. So, in that reference the type casting discussions and type casting would prove to be very useful for you I believe.

Exceptions are from a very different flavour, they for the first time address the question of error situation, exception situations and their handling in the comprehensive manner in terms of the language. Now, we all know that if we write software then the software might not work. So, error handling or being able to debug the programs efficiently is a part and parcel of developers own life, is a part and parcel of a program life cycle.

But unfortunately the C language does not provide in a built-in support in the language for doing handling all these you know unhappy paths, all these error paths and a synchronous, asynchronous the error introduce you to logical reasons the error introduce you to system configurations and all that. So, we took a very detailed look in terms of what C provides, what is the lacunae they are in and in view of that we try to understand the basic semantics of try, throw, catch exception clauses and how does the C++ provide that.

And again exceptions are a mechanism which really completes the whole story of object oriented programming much easier, because exception really make sure that it is no more necessary that you keep talking about doing certain operations and then tracking whether the operations has been correct or not. That if you provide the correct context of try clauses and put appropriate catch handlers, then you can write the whole code without really thinking of the exception situations and fill in the catch handlers for the different exception situations in a completely separate code base, so that being a much better clarity to that.

And at the end, of course, we could spend very little time compared to the depth of this particular topic. We just spent two modules, but this by itself could become a 10 module course is the factor of templates. Templates are completely different concepts in terms of C and C++, because they are certain mechanisms by which you can write a code where one or more variables or parameter or class types are not known at the time of writing the code or at the time of compiling the code. The type could be specified

subsequently when you actually use that function, when you actually use that particular class and we saw examples of this through simple max, swap kind of function and stack kind of data structure. So, templates give a different kind of polymorphism.

So, if you just look at, then templates can be looked at in multiple different aspects. One is, it could look at it from the aspect of polymorphism, so overloading gives us a certain kind polymorphism which is called the static polymorphism. The dynamic binding or what we have ‘return’ here as polymorphism is a dynamic polymorphism, which is basically what happens in a class hierarchy. And template gives you another kind of polymorphism because that is a template polymorphism, because here again you have a single form of a function written in terms of the template of the function or single form of a class, but you can use it for variety of different purposes.

So, when you have a template function instantiation say an implicit instantiation then there is again the same question of binding which we had to answer in case of overloading, which we had to answer in case of dynamic polymorphism, we again have to answer that. So, that is a very different kind of features that comes in templates are code generated; they actually generate code and then compiles that code.

So, if you are able to design templated code in a crisp efficient manner then your basic effort in programming, your basic effort in debugging gets substantially reduces because one code can be used in not only in the multiple context of type through overloading, but you could keep on using it on also for the types that will come in future. So, templates are primarily introduced in C++ for this purpose of generic programming or template meta programming. So, if we look into from that aspect then we will actually see that C++ is a combination of three major paradigms of programming which is procedural, because it includes the whole of C and the better C. So, that gives you a complete set of features and lot of powers in the procedural terms. So, it is always very efficient to write algorithms in C++.

It is known for its object oriented features, so it is called an object oriented programming language as well. So, it strongly supports the object oriented paradigm though it has some lacunae in that and those may be beyond the discussion of this course. But certainly it

cannot do few fundamental things of object orientation like reflection, but what is important is, it also supports a third paradigm which is called a generic programming paradigm, which is a code generation paradigm where you could write type parametrized codes and generate code based on instantiation. So, you can see that C++ basically developed on three different paradigms and therefore, it is very rightly called a multiple paradigm language.

Of course, if you refer back to C, the earlier version of C++, older versions of C++, then you might feel that it is just procedural and then object oriented, but over and over the years, the template features the generic programming features in C++ have been really gaining in strength. And though we have been using primarily and our discussion, we have been talking about C++ 99 which is a 17 year old standard which was a marginally revised in C++ 03, which means the 2003 standard. But subsequently, we have had significant progress in terms of C++ standard. We have two new standards now; one is called C++ 11 which was finally, actually released in 2012; and we have another standard, the most recent standard is C++ 14 which was released last year in 2015.

And we have not discussed as upon any of these features from C++ 11 or C++ 14 which give you a whole lot of very strong additional parameter to strengthen all of these paradigms primarily the object oriented paradigm, and the generic programming paradigm. And brings it lot of strong concepts into the existing features and still maintains complete backward compatibility with the earlier C++ 03 language. So, this is a basically what I would like to just point out that the whole thing that we have covered is just kind of the small part of what is C++ today, and the way it can actually benefit you.

(Refer slide Time: 22:55)

The screenshot shows a presentation slide with a blue header bar containing the title 'What we did not cover?'. On the left side, there is a sidebar with a logo at the top, followed by a vertical list of navigation links: Module 40, Partha Pratim Das, Objectives & Outline, Course Summary, Key Takeaway, Prepare for Examination, and Road Forward Summary. Below the sidebar is a video frame showing a man speaking. The main content area contains a bulleted list of topics:

- **Functors:** Function Objects
- **STL:** Standard Template Library of C++
- **Resource Management:** Smart Pointers, Memory Handling
- **C++ Coding Styles:** How to write good code?
- **Design Patterns:** Reusable Designs
- **Mixing C and C++:** Smart Pointers, Memory Handling
- **Source Code Management:** How to organize files, libraries?
- **C++ Tools:** Analysis, Version Control etc.
- ...

At the bottom of the slide, there is footer text: NPTEL MOOCs Programming in C++, Partha Pratim Das, and a small number '5'.

So, having said that I would also like to point out a few features; now what we have not covered is very difficult to list out because C++ is actually so huge that it is very difficult to say what we have not done. It will not fit into one slide or couple of slides, but here I have just highlighted some of the aspects those are very, very important and you should going forward take initiative to learn them better. So, one certainly is some of the typical programming styles that have emerged based on C++, one is called functors.

Functors are very interesting design, they are called function objects. So, we know functions are what can be called and objects are what can be instantiated. But we have new concepts here where it will say that a function can be instantiated, it could be an object it could have a state. So, it simple a function calls is basically shown in terms of the function operator which is like this. So, a functor object is nothing but a class definition or an object where this function call operator itself is overloaded. So, it does not really as simple as I am saying, but it is almost that. And once you can do that you can have a lot of benefits.

And the next thing that we have not significantly done accept for sparingly using certain data structures like vector and list and stack is the standard template library which is significantly based on the concepts of functor. And it is very critical that you explore and

a slowly get more and more familiar with this standard template library, so that your power in being able to write good C++ code also improves. Couple of other very important aspects which we did not get time for is resource management particularly concepts of smart pointer memory handling and so on. C++ coding style is to how you should be writing codes in C++ certainly not the C way. Design patterns, which tell you, what are the reusable designs? Many of the practical systems will need you to make C with C++ may be there is an existing code base of C you cannot throw all of that away and write C++.

So, you write certain parts in C++ you continue to have the remaining parts in C. So, how you make them is a big question. Management of source code C++ has given features like as you saw namespace and export and all that. And with that you can do a much better C++ source code management and certainly for engineering purposes you need to learn a whole lot of C++ tool. So, there is the list is much, much longer. I have not talked about in this any features which C++ 11 or C++ 14 have introduced these are all aspects of C++ 03, but it will be good overtime, if you can really learn them.

(Refer slide Time: 26:12)

The slide has a blue header with the title "What have we learnt?". On the left, there's a sidebar with navigation links: Module 40, Partha Pratim Das, Objectives & Outcome, Course Summary, Key Take-back, Prepare for Examination, and Road Forward Summary. A small circular video thumbnail of the speaker is on the left. The main content area lists the following bullet points:

- C++ is multi-paradigm
 - Procedural: Better C
 - Object-Oriented: Encapsulation, Inheritance, and Polymorphism
 - Generic: Templates
- Reuse is Key
 - Macros
 - Library functions
 - Function Overloading (Static Polymorphism)
 - Inheritance & Dynamic Polymorphism
 - Templates & STL
 - Design Patterns
- Designing good data types is a key for good programming in C++
- While programming in C++, we should keep an eye on:
 - Efficiency
 - Safety
 - Clarity

A red box at the bottom contains the text "Do not write C-style programs using C++ compiler". The footer of the slide includes the text "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

So, if we try to summarize us to what we have learned. We have learnt the first is the fact that C++ is a multiple paradigm language it is procedural, it is object oriented and it is

generic. So, in future going forward from tomorrow as you do C++ programming, always try to identify which paradigm you are working in, and how do they makes. A second aspect is reuse, the key in C++. If you see the whole gamut of features as we have seen then there C supported things like macros and library functions only this is what you had in C.

And then in C++, we have function overloading, the static polymorphism, the dynamic polymorphism other kinds of reuse options and then when we do templates and as you go into STL, you will find a huge amount of reuse, here because now, you are writing one max function with a parameterised type, and that will work for not only the built in types, but for all future user defined types that you will come across with.

Design patterns are another aspects of reuse where you not only irrelevant limited to reusing code, but you try to reuse the pair designs as well. Certainly the other aspects that you should have learnt well, and is designing good data types is a key for good programming in C++, because C++ is strongly type and everything that you want to do, you want to make a type for that.

And while programming in C++ you should keep an eye on the efficiency because C++ happens to be the most efficient language generic program, I mean general purposes programming language today, even it is more efficient than C. Most experiment show that an equivalent code corresponding to the C code written in the C++ would run anywhere between 50 to 60 to 100 percent faster than the C code. So, efficiency is a key, safety is a key, and certainly all different exceptions and all those add up to the safety features and clarity is a major factor that it should be very clearly understandable.

So, in terms of the take back key I would like to ponder over this points and I would finally, make one caution which just pardon me for saying this, but have been dealing with budding C++ programmers for last possibly 20 years. And what I find is a good tendency that the programmers have is actually use the C++ compiler, but write code in C.

I do not mean using the syntax of C, you can use still use the syntax of C++ you can still use objects and specialization and all that, but the way you do the design is a C style of design. So, I have regularly try to show you the comparison between C style of solving problem and C++ style of solving the problem. Please refer to those please refer to good code solutions, and make sure that when you use C++ you write the code in the C++ style taking full advantage of the multi paradigm language and do not happen to just write a C++ compiler based code with C style.

(Refer slide Time: 29:39)

The screenshot shows a computer desktop with a slide titled "Prepare for Examination" open in a browser window. The slide has a blue header and footer. The main content area contains a list of study tips:

- Watch the Videos
- Revise the Assignments and Solutions
- Practice lots and lots of coding with every feature
- Design and implement complete data types – Complex, Fraction, Vector, Matrix, Polynomial etc.
- Study Books, try examples
 - The C++ Programming Language by Bjarne Stroustrup
 - Effective C++ & More Effective C++ by Scott Meyers

At the bottom of the slide, there is a video thumbnail of a man speaking, with the text "NPTEL MOOCs Programming in C++" and "Partha Pratim Das". The desktop taskbar at the bottom shows various icons, and the system tray indicates a battery level of 70%.

Finally to prepare for your examination these are the routine things; watch the video, revise the assignments and solutions. We will provide explanatory solutions soon, practise lots of lots of coding that is a key of learning this. And certainly design and implement complete data types, I mean we have done complex, these are some of the sample data types which I can where glimpses we have done, but you can practise by doing this data types at length. And if you need to refer to books these are the couple of books that I will recommend you to follow.

(Refer slide Time: 30:17)

The screenshot shows a presentation slide titled "Road Forward" for "Module 40". On the left, there is a sidebar with a logo at the top, followed by a vertical list of navigation links: "Module 40", "Partha Pratim Das", "Objectives & Outline", "Course Summary", "Key Take-back", "Prepare for Examination", "Road Forward", and "Summary". Below the sidebar is a circular profile picture of a man with glasses and a blue shirt. The main content area has a dark blue header with the title "Road Forward". The slide contains a bulleted list of study recommendations and a list of recommended books:

- Learn the topics not covered
- Breathe programming – regularly code and implement systems
- Read lots and lots of programs by good coders
- Learn Python / Java
- Study **Object Oriented Analysis and Design**
- Study **Unified Modeling Language**
- Study **Software Engineering**
- Study Books
 - The C++ Programming Language by Bjarne Stroustrup
 - Effective C++ & More Effective C++ by Scott Meyers
 - Exceptional C++ & More Exceptional C++ by Herb Sutter
 - Modern C++ Design by Andrei Alexandrescu
 - Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, & John Vlissides
 - Learning UML 2.0 – A Pragmatic Introduction to UML by Russ Miles & Kim Hamilton (O'Reilly)

At the bottom of the slide, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das". There is also a small number "8" in the bottom right corner.

Going forward, beyond this course, I could tell you lot of things that you could do, you could learn the topics that were covered. But the core things you must breathe programming regularly code and implement systems, read lots of code that these two are very, very important to learn C++. And beyond that these are all the futures studies that you can do. It is good to learn other object-oriented languages to understand one language better. So, learn python, learn java. If you get an opportunity and go through object-oriented analysis and design UML for modelling the system software engineering and here are some of the very good books which deal with C++ or some of this related topics.

(Refer slide Time: 31:03)

Module Summary

- Course on C++ concluded

Module 40
Partha Pratim Das
Objectives & Outline
Course Summary
Key Takeaway
Prepare for Examination
Road Forward Summary

NPTEL MOOCs Programming in C++
Partha Pratim Das

So, with that, I would like to close and summarize simply saying that the course on C++ is concluded. Wish you all the best for your examination and beyond that to become a very efficient, proficient and prolific programmer in C++.

**THIS BOOK
IS NOT FOR
SALE
NOR COMMERCIAL USE**



(044) 2257 5905/08



nptel.ac.in



swayam.gov.in